

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

区块链技术 进阶与实战

蔡亮 李启雷 梁秀波 ● 著



- 专注介绍区块链核心原理和应用技术
- 详细解读区块链平台以太坊和HyperLedger
- 注重实战，全书包含5个完整实际项目案例



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

蔡亮，博士，副教授，浙江大学计算机学院软件工程系主任，浙江省重大科技专项专家。主要从事区块链、云计算、网络安全、可信计算和金融业务处理的研究，在国家级核心期刊和国际会议上发表了数十篇论文。参与了多项国家级或省部级科研项目，如国防军工预研基金项目、国家创新基金项目、863项目等。获得教育部科技进步一等奖、浙江省科技进步一等奖和三等奖。

李启雷，博士，讲师，杭州趣链科技有限公司首席技术官。主要从事体感人机交互、区块链和移动互联网技术等方面的研究与开发。作为核心研究人员参与国家863计划和国家科技支撑计划，在国内外学术期刊发表论文九篇，获得国家发明专利一项、软件著作权一项。

梁秀波，博士，副研究员，杭州趣链科技有限公司副总经理。主要从事机器学习、区块链、数字娱乐和移动互联网等方面的研究与开发，曾赴法国进行为期一年的访问研究。作为核心研究人员参与国家级和省部级科研项目近十项，主持企事业单位委托项目二十余项。已发表论文十余篇，获得国家发明专利三项。

TURING 图灵原创

区块链技术 进阶与实战

蔡亮 李启雷 梁秀波 ● 著

人民邮电出版社
北京

图书在版编目(CIP)数据

区块链技术进阶与实战 / 蔡亮, 李启雷, 梁秀波著.

— 北京: 人民邮电出版社, 2018.4

(图灵原创)

ISBN 978-7-115-47179-6

I. ①区… II. ①蔡… ②李… ③梁… III. ①电子商务—支付方式 IV. ①F713.361.3

中国版本图书馆CIP数据核字(2017)第286209号

内 容 提 要

本书从实战的角度出发, 结合实际应用开发场景, 对区块链技术进行了全面介绍和剖析。全书共分为四个部分: 第一部分对区块链进行了全景分析, 介绍了其概念、历史、技术流派和典型应用场景, 并给出了当前区块链产业生态图谱; 第二部分对知名开源区块链平台以太坊和 Hyperledger 进行了详细解读, 并介绍了如何基于这两个平台进行区块链应用开发; 第三部分以自主可控联盟区块链 Hyperchain 为例, 分析了企业级区块链平台的核心技术, 介绍了基于 Hyperchain 的企业级区块链应用开发技术; 第四部分介绍了多个区块链实际应用项目案例, 并对开发过程和关键代码进行了详细分析。本书不仅全面深入, 而且注重实战, 非常适合区块链开发人员由浅入深地学习区块链技术。

本书适用于对区块链技术感兴趣的程序员、架构师和高校学生, 可作为高校教材和培训资料。

◆ 著 蔡 亮 李启雷 梁秀波

责任编辑 张 霞

责任印制 周昇亮

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京鑫正大印刷有限公司印刷

◆ 开本: 800×1000 1/16

印张: 18.75

字数: 443千字

2018年4月第1版

印数: 1—4 000册

2018年4月北京第1次印刷

定价: 69.00元

读者服务热线: (010)51095186转600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京东工商广登字 20170147号

序 一

打造自主可控联盟区块链，开创金融科技发展新天地

近年来，以比特币为代表的“数字加密货币”在世界范围内广为流行，其价格经历了数轮爆发式增长，对现有金融体系产生了一定的冲击，同时带来了新的机遇和挑战。区块链作为“数字加密货币”系统的底层支撑技术，展现出了巨大的潜在应用价值，将在金融、贸易、物流、征信、物联网、共享经济等诸多领域引发技术革新浪潮。截至 2017 年底，全球已有 20 多个国家投资了区块链技术，区块链领域的风险投资超过了 20 亿美元，区块链相关专利数量也已超过 3000 个。我国政府对区块链技术的发展也非常重视，2016 年 12 月，国务院将区块链列入“十三五”国家信息化规划。总体上，当前区块链技术尚处于起步阶段，国内外发展差距并不大。为了避免在该重要新兴技术领域出现类似操作系统、数据库等底层关键技术被国外巨头垄断的局面，研发一套完全自主可控的区块链底层平台意义重大。

区块链技术本质上是一种分布式账簿数据库，它利用块链式数据结构来验证与存储数据，基于分布式共识算法来生成和更新数据，并通过密码学的方式保证数据传输和访问的安全。从功能层面上来看，区块链记录不可篡改，无需第三方可信中介，天然适合多个机构在区块链网络中相互监督并实时对账，通过智能合约大大提高了经济活动与契约的自动化程度。按其组织形态，区块链平台可分为公有链、联盟链和私有链。公有链是非许可链，具有完全去中心化的特性，但存在共识效率极低、缺乏权限控制与隐私保护等问题，除了“数字加密货币”之外，很难应用于其他领域。联盟链是许可链，需经过一定的权限许可方能加入网络。私有链也属于许可链，其许可权掌握在单一机构手中。从时间维度来看，区块链技术自诞生以来发生了三次重要的技术演进。第一次发生在 2009 年，代表平台为比特币，首次验证了无中心机构的“数字货币”的可行性，但其交易频率极低，仅为每秒几笔交易。第二次发生在 2013 年，代表平台为以太坊，首次在区块链平台中增加了可编程特性，从而大大拓展了区块链的应用范围，但其交易频率仍然不高，仅为每秒几百笔。第三次发生在 2015 年，代表平台为 Hyperledger Fabric 和 Hyperchain，首次在区块链平台中加入权限控制和隐私保护，并将交易频率提高到了每秒几千笔甚至到上万笔。

联盟区块链具有高效共识、智能合约、多级加密、权限控制、隐私保护等特性，辅以可视监控、动态配置等功能，主要面向企业级应用场景，是区块链发展的最新形态，在中国有极其广泛的应用价值，其核心优势主要有三点：(1) 从监管角度看，联盟区块链可以通过 CA 认证准入、制定监管规则合约等方式为监管提供便利；(2) 商业机构及用户对账户和部分交易信息有隐私保护的需求，联盟区块链可以通过加密、分区等方式实现隐私保护；(3) 从商业应用角度来看，交易吞吐量和时延是企业最关心的交易性能指标，联盟区块链通过共识算法的创新使交易效率得到很大提升。但是，一些商业需求的场景对联盟区块链提出了更高的技术要求，例如：(1) 高性能，如何在多个节点之间高效地达成共识，如何有效地提升智能合约的执行效率；(2) 高可用，应允许在不宕机的情况下加入新节点，并可在节点重启之后快速恢复；(3) 安全隐私，如何设计权限控制机制使之符合国家标准，并能有效地保护隐私数据；(4) 可编程，应提供图灵完备的、安全的智能合约引擎，可支持多语言的、复杂的智能合约。然而，当前主流的开源区块链平台（如以太坊、Hyperledger Fabric 等）尚未达到上述技术要求。

在杨小虎研究员、蔡亮副教授的领导下，浙江大学超大规模信息系统研究中心对联盟区块链的核心技术开展了多层次的研究工作。杭州趣链科技有限公司的核心骨干均来自于浙江大学超大规模信息系统研究中心，公司研发的 Hyperchain 联盟区块链平台在高性能、高可用、安全隐私和可编程技术方面取得突破，支持了国内首个接入银行核心系统的区块链项目的落地和稳定运行。目前 Hyperchain 平台已在金融等众多领域得到了实际应用。

本书对以太坊、Hyperledger Fabric 和 Hyperchain 的技术特点及内核代码进行了详细的分析，对各平台的应用开发技术进行了介绍。相信本书对区块链技术的爱好者和区块链行业的从业者会有很好的参考价值。

陈纯

中国工程院院士，浙江大学计算机科学与技术学院教授，
曾任浙江大学软件学院院长和浙江大学计算机软件研究所所长

序 二

利用区块链构建新型多方业务协作平台

很荣幸受邀为本书撰序，也很高兴看到这样一本全面、系统、综合介绍区块链技术的图书问世。

随着一系列以比特币为代表的“虚拟货币”大热，区块链这一底层支撑技术浮出水面，受到广泛关注。相关从业人员逐渐发现该技术的应用场景不仅是产生一种新型的“数字货币”，而是解决万物互联的产品交换问题，改变商品交易模式，进而影响和改变未来的经济和金融形态。甚至认为它像 TCP/IP 协议一样，将对互联网的发展产生颠覆性的作用，在未来万物互联的世界里，对人类社会产生重大影响，彻底改变社会生产方式和人们的生活方式。以 TCP/IP 协议为基础的第一代互联网只解决了信息传输的效率问题，却未解决信任问题。在涉及多方协作的业务场景时，除了要建设信息系统、建立中介机构以完成信息传输和交换外，还需要额外采取一系列的措施来解决各方信任问题，这大大增加了各方沟通和协作的成本。区块链技术通过建立多节点参与共识的、难以篡改的分布式总账，使得利益相关方可以从技术层面上共组一个网络、共记一套账本，因而可以大大降低业务协作过程中的沟通和人力成本。

对新型多方业务协作系统而言，区块链的价值主要体现在 4 个方面：(1) 降低系统对接复杂性，跨系统数据交互与路由下沉到区块链层，降低应用开发的难度和成本，并提高开发效率；(2) 提高数字资产流动性，通过资产上链，提高流动性，实现价值传输；(3) 实现全流程监控，智能合约记录信息和状态的流转，数据难以篡改，实现信息的全流程可监控；(4) 多方可信合作，使产业链可信合作成为可能，符合轻资产运行需求。

中国外汇交易中心多年致力于金融科技（FinTech）领域的前沿技术研究，在区块链、人工智能、分布式架构、微服务和软件形式化等方面均有探索，对于区块链和分布式账本技术的研究，形成了符合中心技术规划和技术发展路线的区块链架构方案，并以国密算法、共识协议、智能合约等关键技术为突破口，明确了对通用区块链技术的改进方案，取得了丰硕的研究成果。交易中心与浙江大学杨小虎研究员、蔡亮副教授的研究团队在金融科技领域有多年的合作研究基础，在

区块链方面从 2016 年开始与杭州趣链科技有限公司展开合作，现已形成一定的技术储备。杭州趣链科技有限公司研发的 Hyperchain 平台具有性能优异、安全可靠、监控可视化、支持合约无缝升级和数据存储横向扩展等特性，是国内非常具有竞争力的、自主可控的企业级区块链底层平台，是基于区块链搭建新一代价值传输和业务协作平台的理想选择。

除了金融领域，区块链技术在贸易、物流、征信、物联网、社会公益等领域也得到了越来越多的关注和应用，越来越多的专业人员开始从事区块链平台研究和应用开发工作，但当前市面上介绍区块链开发技术的图书却不多，可指导读者动手实践的书更是凤毛麟角。本书基于浙江大学和杭州趣链科技有限公司多年的区块链技术研发经验，对知名的开源区块链平台以太坊和 Hyperledger Fabric 以及自研的 Hyperchain 平台展开了深入剖析，在讲解平台功能的过程中，穿插说明了区块链的关键算法和核心原理，并配有各个平台的开发指南和项目案例，深入浅出地介绍了当前主流的区块链开发技术。相信本书可帮助读者更深刻地理解区块链技术原理，并有效地提升区块链技术开发能力。

许再越

中国外汇交易中心副总裁

前言

区块链技术是金融科技领域乃至整个 IT 领域的重大技术创新。该技术本质上是以数据加密、时间戳和分布式共识算法等基础技术为依托,实现链式存储、智能合约和隐私保护等高级功能的分布式账本技术。该技术通过区块链网络节点之间的相互验证、监督和数据备份,从技术层面上保证在链式账本中所存储的数据无法被恶意篡改,特别适合用于解决多方业务协作场景中为维护信用而导致的成本居高不下的问题。

区块链技术起源于比特币,是“数字加密货币”的底层支撑技术。自 2009 年诞生以来,比特币系统已在无中心维护机构的情况下稳定运行达 8 年之久,单个比特币的价格也经过了数轮暴涨。随着比特币的流行,数以百计的“数字加密货币”快速涌现。近年来,人们发现“数字加密货币”背后的区块链技术可能具有发挥更大价值的潜力,将来可用于极为广泛的业务场景。许多专家认为,区块链技术可用于解决新一代互联网去中心化的价值交换问题,即网络传输的信用问题。

用于“数字加密货币”的区块链技术只能实现交易转账等基础功能,被认为是 1.0 版本的区块链技术。要想将区块链应用于“数字加密货币”之外的广泛场景,必须对该技术加以改进。2014 年,以太坊应运而生。该平台通过支持智能合约将业务逻辑的设计和控制权转移到了平台用户手中,允许通过编写合约代码来满足各种复杂的业务场景的需求。该平台作为区块链 2.0 版本的典型代表,广受赞誉和追捧。

然而,以太坊平台并不是一个完美的区块链平台,还存在共识效率低下、隐私保护缺乏、大规模存储困难和信息难以监管等问题,无法应用于大规模的企业级信息系统。针对这些问题,一些企业级的区块链平台诞生了,其中的典型代表是 IBM 支持的 Hyperledger Fabric 和趣链科技的 Hyperchain。通过高效共识、多级加密、权限控制、可视监控、动态配置等技术,企业级区块链平台为区块链技术打开了更为广泛的应用空间。

本书是一本介绍区块链核心原理和应用技术的专业图书,与当前市场上的绝大部分区块链图书不同,不会天马行空地设想各种短期无法落地的应用场景,而是专注于介绍技术“干货”和实战,读者通过本书的学习即可上手使用当下最受欢迎的区块链平台,参考本书实例即可快速开发自己的第一个区块链应用,实战性非常强。

本书结构

本书共分为四个部分。

第一部分介绍区块链的基础知识,使读者快速对区块链技术有一个整体认识。本部分包含了 1 章内容,即第 1 章,对区块链技术的发展情况进行了全面分析,介绍了其概念、历史、技术流派、关键技术和典型应用场景,通过对主流平台进行对比分析,给出当前区块链产业生态图谱。

第二部分对知名开源区块链平台以太坊和 Hyperledger 进行详细解读,并介绍如何基于这两个平台进行区块链应用开发。本部分包含了 4 章的内容。

第 2 章对以太坊的发展历史、基本概念、客户端、账户管理及以太坊网络等基础知识进行了介绍,并对以太坊共识机制、虚拟机、数据存储和加密算法等以太坊关键模块的核心原理进行了剖析,详细介绍了以太坊智能合约的编写、部署、测试与执行,最后对以太坊发展过程中的重大事件和目前存在的主要问题进行了分析探讨。

第 3 章首先介绍了如何搭建以太坊的开发环境,包括 Go 语言环境、Node.js 和 npm 的配置、Solc 编译器的安装,以及如何使用以太坊 geth 客户端搭建私有链;接着讲解了以太坊智能合约开发的集成开发环境,包括 Mix IDE 和在线浏览器编译器;然后讲述了 JSON RPC 和 JavaScript API 两种以太坊编程接口,通过这两种接口可以实现和以太坊底层的交互,实现合约方法的调用;随后讲述了目前主流的以太坊开发框架与流程,包括 Metero、Truffle 和商业化开发中的分层可扩展开发流程;最后给出了第一个较为完整的以太坊应用开发实例。

第 4 章对 Hyperledger Fabric 进行了深入解读,有助于读者深入理解 Fabric 的底层实现原理。首先,介绍了 Hyperledger 及其子项目的发展现状及管理模式,重点介绍了 Hyperledger Fabric。之后,对 Hyperledger Fabric 架构进行深入分析,从成员服务、区块链服务以及合约代码服务三个方面探讨 Hyperledger Fabric 的架构组成与特点,给出了 Fabric 架构设计和模块组件。然后,给出了 Chaincode 代码结构、调用方式和执行流程。最后,对交易背书流程展开了详细分析。

第 5 章主要介绍如何在 Hyperledger Fabric 平台上开发区块链应用,首先讲述了 Hyperledger Fabric 的开发运行环境的搭建过程,然后给出了 Chaincode 开发和部署流程,最后介绍了 CLI 应用接口和 SDK 接口,并通过实例说明了如何基于这两种接口开发 Hyperledger Fabric 的区块链应用。

第三部分以自主可控联盟区块链 Hyperchain 为例对企业级区块链平台的核心技术进行分析,并介绍了基于 Hyperchain 的企业级区块链应用开发技术。本部分包含了 2 章的内容。

第 6 章以企业级区块链平台 Hyperchain 为例,介绍了构成企业级区块链平台的核心组件的实现原理。企业级区块链同公有链和私有链不同,它直接面对企业级应用的需求,对区块链系统的安全性、灵活性以及性能都有着更加严格的要求。Hyperchain 企业级区块链平台在优化传统 PBFT 的基础上设计实现了灵活、高效、稳定的共识算法 RBFT,在智能合约的支持上选择了支持开源领域活跃的 Solidity 语言,对其执行虚拟机进行了系统层面的优化,并通过对交易、交易链路、应用开发包等多层面进行加密处理,加强企业级区块链的安全等级。此外,Hyperchain 还设计实

现了支持系统监控、合约编写、合约编译等多功能的企业级区块链管控平台。

第 7 章主要介绍了 Hyperchain 区块链上应用开发的相关内容。首先,从交易调用、合约管理以及区块查询几个方面介绍了 Hyperchain 平台对外提供的主要接口;其次,从 Hyperchain 集群的配置、部署和运行等方面介绍了如何搭建一个可运行的企业级区块链系统 Hyperchain;最后以模拟银行为例,介绍了如何在 Hyperchain 平台上进行智能合约应用的开发。

第四部分介绍了多个区块链实际应用项目案例,并对其开发过程和关键代码进行了详细分析。本部分包含了 2 章的内容。

第 8 章介绍了两个基于以太坊的实际项目案例,每个案例的介绍均包括项目简介、系统功能分析、系统总体设计、智能合约设计、系统实现和部署等部分,并通过相关下载链接提供了这些案例完整的源代码。基于前面章节所学习的以太坊基础知识和开发技术,读者可对照本章的内容,一步一步地动手实践,在实战过程中更好地理解相关概念和技术,从而为自己基于以太坊构建区块链应用项目打好基础。

第 9 章介绍了两个基于 Hyperchain 的企业级区块链应用项目案例,每个案例的介绍均包括项目简介、系统功能分析、系统总体设计、智能合约设计、系统实现和部署等部分。可以看到,利用 Hyperchain 可以构建功能完备、技术领先、符合企业级要求的区块链应用。读者可对照本章内容,通过 Hyperchain 提供的完善的开发接口,对区块链应用开发进行深入的学习和实践。

示例代码与勘误

本书第 8 章和第 9 章中的所有项目案例代码已上传至 GitHub 供开发者下载: <https://github.com/Blockchain-book>, 后续若增加新的项目案例,也会同步更新到该仓库中。

开发者可以使用 Git 版本控制 clone 项目到本地或直接下载代码,目前默认的检出分支为 master 分支。如果对某项目有疑问或意见,可以在该项目中提交 Issue。如果想要接收某项目的更新邮件提醒,请点击 Watch;如果要持续关注某项目,请点击 Star;如果要复制代码到自己的账户,请点击 Fork。

由于作者时间和水平有限,本书难免会存在一些纰漏和错误,欢迎广大读者批评指正。勘误请提交至图灵社区本书主页 www.it-ebooks.com.cn/book/2434,或发送至作者邮箱: liangxiubo@hyperchain.cn。对于读者发现的问题,我们将在本书后续印次和版本中加以改正。

开发者平台及更多技术支持

为了进一步降低区块链技术使用门槛,让更多的区块链开发者、爱好者以及正在尝试接入区块链技术的企业能够快捷地开发区块链应用,趣链科技于 2017 年 9 月 14 日正式上线了基于联盟链的“开发者平台”。基于该平台,用户可以更方便地创建、发布和使用多中心化的应用程序。

通过平台提供的在线智能合约编辑器,用户可便捷、准确地编写智能合约程序;通过平台提供的区块链浏览器,用户可方便地获取链上区块信息、区块链节点状态、节点维护方信息等。欢迎广大区块链相关从业人员访问体验,开发者平台网址为:<https://dev.hyperchain.cn/>。

如需获得更多关于区块链技术的技术动态和趣链科技的技术支持,可扫描如下二维码关注微信公众号。



致谢

作为区块链技术人员,能够编写一本技术性和实践性非常强的区块链图书,我们感到非常荣幸。在此向所有给我们提供指导、支持和鼓励的朋友表示衷心的感谢。

感谢浙江大学计算机科学与技术学院和软件学院为我们提供的优良条件和各种便利,感谢陈纯院士、杨小虎研究员一直以来的关怀和支持,使得本书得以顺利完稿。

感谢杭州趣链科技有限公司全体人员的大力支持,特别感谢李伟博士、邱炜伟博士后、尹可挺博士为本书成稿所给予的鼎力支持,感谢汪小益、黄方蕾、戎佳磊、陈宇峰、吴发翔、吴琛、胡为、宋家锦、郭威、李超等对书稿材料汇编所做出的突出贡献,感谢刘耀傲、胡麦芳、卓海振、钟蔚蔚、孙琪、赵科、黄志胜等对书稿校阅所付出的时间和汗水。

感谢万达网络科技集团先进技术研究 centers 副总经理季宙栋、区块链资深研究员张梦航对本书第4章和第5章内容的有益补充。

感谢人民邮电出版社图灵公司的编辑们,是他们不辞辛苦、仔细严谨的审阅和校对工作为本书的顺利出版提供了有力保障。

蔡亮 李启雷 梁秀波

2017年9月于浙江杭州

目 录

第一部分 区块链基础

第1章 区块链基础入门	2
1.1 区块链基础知识	2
1.1.1 从比特币到区块链	2
1.1.2 区块链定义	3
1.1.3 区块链相关概念	4
1.1.4 区块链分类	6
1.2 区块链发展历程	8
1.2.1 技术起源	8
1.2.2 区块链1.0——“数字货币”	9
1.2.3 区块链2.0——智能合约	9
1.2.4 区块链3.0——超越货币、 经济和市场	10
1.3 区块链关键技术	10
1.3.1 基础模型	10
1.3.2 数据层	11
1.3.3 网络层	16
1.3.4 共识层	17
1.3.5 激励层	19
1.3.6 合约层	20
1.4 区块链产业现状	21
1.4.1 区块链发展态势	21
1.4.2 区块链政府规划	22
1.4.3 区块链生态图谱	25
1.5 区块链应用场景	26
1.5.1 数字票据	26
1.5.2 供应链金融	27
1.5.3 应收账款	27
1.5.4 数据交易	27
1.5.5 债券交易	28

1.5.6 大宗交易	28
1.5.7 其他场景	28
1.6 区块链主流平台	29
1.7 本章小结	31

第二部分 开源区块链平台

第2章 以太坊深入解读	36
2.1 以太坊基础入门	36
2.1.1 以太坊发展历史	36
2.1.2 以太坊基本概念	37
2.1.3 以太坊客户端	39
2.1.4 以太坊账户管理	43
2.1.5 以太坊网络	46
2.2 以太坊核心原理	46
2.2.1 以太坊共识机制	47
2.2.2 以太坊虚拟机	49
2.2.3 以太坊数据存储	51
2.2.4 以太坊加密算法	53
2.3 以太坊智能合约	53
2.3.1 智能合约与Solidity简介	54
2.3.2 智能合约的编写与部署	56
2.3.3 智能合约测试与执行	67
2.3.4 智能合约实例分析	71
2.4 以太坊重大事件与现存问题	74
2.4.1 The DAO攻击事件	74
2.4.2 以太坊现存问题	75
2.5 本章小结	77
第3章 以太坊应用开发基础	78
3.1 以太坊开发环境搭建	78

3.1.1	配置以太坊环境	78
3.1.2	搭建以太坊私有链	80
3.2	以太坊集成开发环境	83
3.2.1	Mix IDE	83
3.2.2	Solidity 在线实时编译器 IDE	86
3.3	以太坊编程接口	89
3.3.1	JSON RPC	89
3.3.2	JavaScript API	94
3.4	DApp 开发框架与流程	97
3.4.1	Meteor	97
3.4.2	Truffle	100
3.4.3	分层可扩展开发流程	102
3.5	第一个以太坊应用	104
3.5.1	优化 MetaCoin 应用	104
3.5.2	MetaCoin 代码详解	106
3.5.3	MetaCoin 应用运行	109
3.6	本章小结	111
第 4 章	Hyperledger Fabric 深入解读	112
4.1	项目介绍	112
4.1.1	项目背景	112
4.1.2	项目简介	113
4.2	Fabric 简介	115
4.3	核心概念	116
4.4	架构详解	118
4.4.1	架构解读	118
4.4.2	成员服务	120
4.4.3	区块链服务	124
4.4.4	合约代码服务	127
4.5	合约代码分析	128
4.5.1	合约代码概述	128
4.5.2	合约代码结构	129
4.5.3	CLI 命令行调用	131
4.5.4	链码执行泳道图	132
4.6	交易流程	133
4.6.1	通用流程	133
4.6.2	流程详解	135
4.6.3	背书策略	138
4.7	本章小结	139

第 5 章	Hyperledger Fabric 应用开发基础	140
5.1	环境部署	140
5.1.1	软件下载与安装	140
5.1.2	开发环境搭建	142
5.1.3	Go 和 Docker	144
5.2	链码开发指南	147
5.2.1	实现智能合约的接口	147
5.2.2	智能合约的依赖关系	148
5.2.3	智能合约的数据格式	148
5.2.4	智能合约的接口解析	149
5.2.5	智能合约案例代码分析	149
5.3	CLI 应用实例	152
5.3.1	CLI 介绍	152
5.3.2	CLI 应用开发	153
5.4	SDK 应用实例	163
5.4.1	SDK 介绍	163
5.4.2	SDK 应用开发	165
5.5	本章小结	169

第三部分 企业级区块链平台 Hyperchain

第 6 章	企业级区块链平台核心原理剖析	172
6.1	Hyperchain 整体架构	172
6.2	共识算法	174
6.2.1	RBFT 概述	174
6.2.2	RBFT 常规流程	174
6.2.3	RBFT 视图更换	176
6.2.4	RBFT 自动恢复	177
6.2.5	RBFT 节点增删	178
6.3	智能合约	179
6.3.1	智能合约执行引擎	180
6.3.2	HyperVM 设计原理	180
6.3.3	HyperVM 执行流程	181
6.4	账本数据存储机制	183
6.4.1	区块链	183
6.4.2	合约状态	185
6.4.3	Merkle 树	186
6.5	安全与隐私机制	189

第8章 以太坊应用实战案例详解214

8.1	基于以太坊的通用积分系统案例 分析	214
8.1.1	项目简介	214
8.1.2	系统功能分析	215

8.1.3	系统总体设计	216
8.1.4	智能合约设计	218
8.1.5	系统实现	224
8.1.6	系统部署	233
8.2	基于以太坊的电子优惠券系统案例 分析	235
8.2.1	项目简介	235
8.2.2	系统功能分析	236
8.2.3	系统总体设计	237
8.2.4	智能合约设计	239
8.2.5	系统实现与部署	246
8.3	本章小结	250

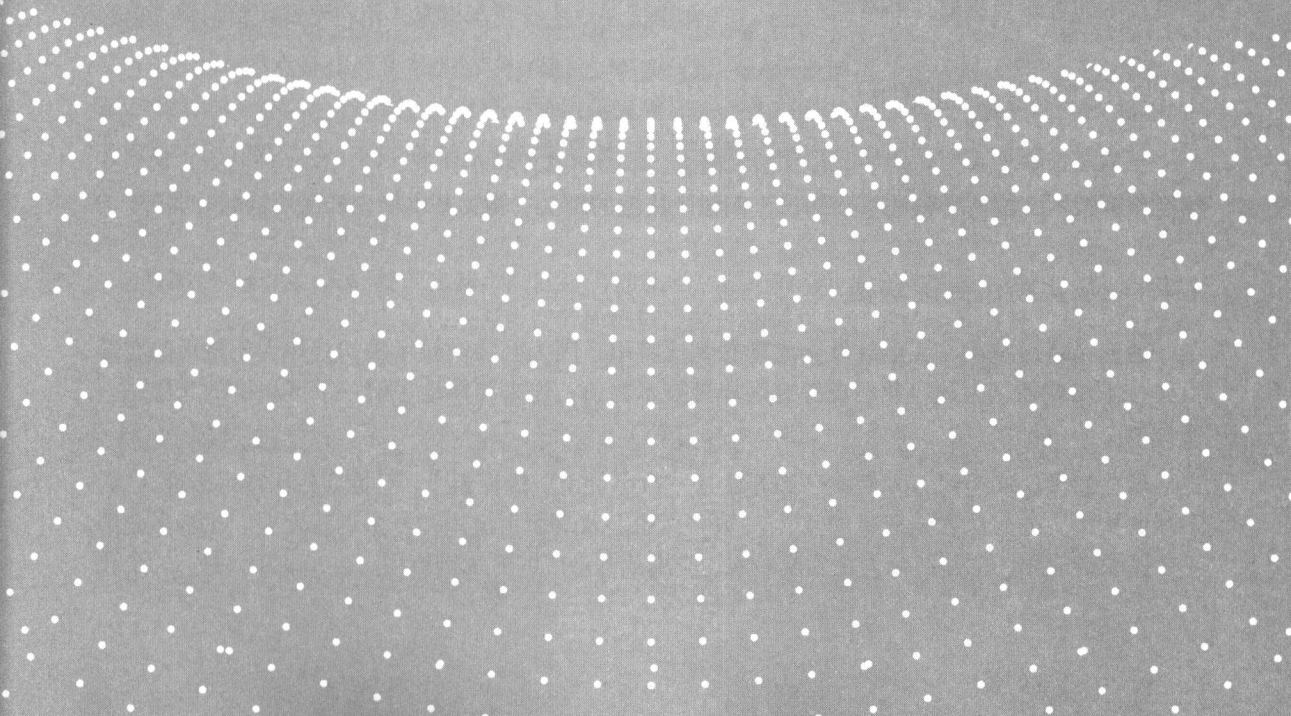
详解 251

9.1	基于 Hyperchain 的数字票据系统	
	案例分析.....	251
9.1.1	项目简介.....	251
9.1.2	系统功能分析	252
9.1.3	系统总体设计	253
9.1.4	智能合约设计	256
9.1.5	系统实现与部署.....	261
9.2	基于 Hyperchain 的出行打车平台	
	案例分析.....	263
9.2.1	项目简介.....	263
9.2.2	系统功能分析	264
9.2.3	系统总体设计	267
9.2.4	智能合约设计	269
9.2.5	系统实现与部署.....	284
9.3	本章小结.....	286

第一部分

区块链基础

□ 第1章 区块链基础入门





区块链技术最初源自于中本聪（Satoshi Nakamoto）2008年提出的比特币（Bitcoin），其去中心化、开放性、信息不可篡改等特性将很可能对金融、服务等一系列行业带来颠覆性的影响。2016年1月，中国人民银行在北京召开“数字货币”研讨会，探讨采用区块链技术发行“虚拟货币”的可行性^[1]，这一消息迅速在各大主流媒体和社区传播和热炒，于是“区块链”这个带着些神秘色彩的名词突然间成为热议的话题，接踵而来的是区块链技术在国内外迅速升温，越来越多的区块链初创公司和相关研究机构小组相继成立，这带动了区块链技术高速发展，使其成为近年来最具革命性的新兴技术之一，甚至被认为是继大型机、个人电脑、互联网、移动/社交网络之后计算范式的第五次颠覆式创新，同时它还被誉作为人类信用进化史上继血缘信用、贵金属信用、纸币信用之后的第四个信用里程碑^[2]。

本章将对区块链技术进行全面剖析，从区块链的基础知识、发展历程、关键技术、产业现状、场景模式和主流平台等方面进行全景分析，使读者对区块链技术有一个整体而直观的认识，为区块链技术的进阶与实战打好基础。

1.1 区块链基础知识

学习一项新技术，必始于了解其基本概念。本节将从比特币讲起，引出区块链技术，然后介绍区块链技术入门所必备的基础知识，例如区块链的定义、相关基本概念和区块链的分类等。

1.1.1 从比特币到区块链

谈到区块链技术，人们往往会先联想到比特币，因为区块链技术最初是作为比特币的底层框架技术出现的。因此，我们在探究区块链技术之前，先来简单地了解一下区块链的起源——比特币。

早在20世纪80年代，人们就已经开始了“数字货币”的探索^[3]。但是直到比特币出现，“数字加密货币”的想法才变成了现实^[4]，“数字货币”及其衍生应用才开始迅猛发展。比特币是第一个区块链应用，也是迄今为止规模最大、应用范围最广的区块链应用。在2008年11月，一个化

名为中本聪的人在一篇“比特币：一种点对点的电子现金系统”论文中，描述了一种如何建立一套全新的、去中心化的点对点交易系统的方法^[5]，并将他在论文中提出的理念付诸实践，着手开发比特币的发行、交易和账户管理系统。2009年1月3日，比特币系统正式开始运行，比特币的第一个区块（也称“创世区块”）诞生了。不久后的2009年1月12日，中本聪通过比特币系统发送了10个比特币给密码学家哈尔·芬尼（Hal Finney），这是比特币系统自上线以来完成的第一笔交易。尽管充满了争议，但从技术角度来说，比特币是“数字货币”历史上一次了不起的创新。自2009年上线以来，它在没有任何中心机构运维参与的情况下，在全球范围内运行了8年多的时间，最大支持过单笔1.5亿美元的交易。而根据blockchain.info统计，截至2017年2月22日，平均每天有约总价值2.3亿元的28万笔交易写入比特币区块链账本中，截至2017年7月，比特币系统已累计生成超过47万个区块。专家预测，到2019年将会有500万比特币用户，这个不断增长的群体可以在在线商城中购买比特币，并把这些比特币放入他们的“数字钱包”，用来购买商品和服务^[6]。

与传统货币和在比特币诞生之前的“数字货币”相比，比特币最大的不同是不依赖于任何中心化机构，而是仅仅依赖于其系统中完全透明的数学原理——加密和共识算法。这就是技术创新所带来的美好，人们不再需要为了信任某个机构而进行一系列的保护措施。这是比特币和区块链技术受到如此多关注和追捧的最主要原因。

比特币作为一种基于区块链技术创造出的“虚拟数字货币”，旨在解决之前的“数字货币”所存在的以下几个问题^[7]：

- 发行机构控制货币的发行以及相关政策，可以决定一切；
- 以前的“数字货币”都无法做到匿名化交易；
- 货币自身的价值无法得到保证；
- 所持货币对于持币人来说不具备完全的安全性。

当前的银行系统作为货币的第三方机构，确实可以有代价地解决上面的几个问题，但是如果把交易范围扩大到全球范围，又有哪一所银行能确保它在全球都是可以信任的呢？于是，就有人提出是否可以设计出一套分布式的数据库系统，它在全球范围内都可访问，并完全中立、公正、安全。很多研究者都努力探索并提出了一些解决方案，但都由于种种原因未能真正被社会接纳，而比特币实现了这样的分布式账本技术。

从2014年开始，人们发现比特币的底层支撑技术区块链具有巨大的潜在应用价值，这正式引发了分布式账本（Distributed Ledger）技术的革新浪潮。随着探索者们的不断创新，区块链技术已经脱胎于比特币，在金融、贸易、物流、征信、物联网、共享经济等诸多领域崭露头角。

1.1.2 区块链定义

区块链技术本质上是一个去中心化的数据库，它是比特币的核心技术与基础架构，是分布式数据存储、点对点传输、共识机制、加密算法等计算机技术的新型应用模式^[8]。狭义来讲，区块链是一种按照时间顺序将数据区块以顺序相连的方式组合成的一种链式数据结构，并以密码学方

式保证的不可篡改、不可伪造的分布式账本。广义来讲,区块链技术是利用块链式数据结构来验证与存储数据、利用分布式节点共识算法来生成和更新数据、利用密码学方式保证数据传输和访问的安全、利用由自动化脚本代码组成的智能合约来编程和操作数据的一种全新的分布式基础架构与计算范式^[9]。

区块链上存储的数据需由全网节点共同维护,可以在缺乏信任的节点之间有效地传递价值。相比现有的数据库技术,区块链具有以下技术特征。

1. 块链式数据结构

区块链利用块链式数据结构来验证和存储数据,通过上文对区块链基本概念的介绍可以知道,每个区块打包记录了一段时间内发生的交易是对当前账本的一次共识,并且通过记录上一个区块的哈希值进行关联,从而形成块链式的数据结构。

2. 分布式共识算法

区块链系统利用分布式共识算法来生成和更新数据,从技术层面杜绝了非法篡改数据的可能性,从而取代了传统应用中保证信任和交易安全的第三方中介机构,降低了为维护信用而造成的时间成本、人力成本和资源耗用。

3. 密码学方式

区块链系统利用密码学的方式保证数据传输和访问的安全。存储在区块链上的交易信息是公开的,但账户的身份信息是高度加密的。区块链系统集成了对称加密、非对称加密及哈希算法的优点,并使用数字签名技术来保证交易的安全。

区块链系统的以上技术特征决定了其应用具有如下功能特征。

1. 多中心

不同于传统应用的中心化数据管理,区块链网络中有多个机构进行相互监督并实时对账,从而避免了单一记账人造假的可能性,提高了数据的安全性。

2. 自动化

区块链系统中的智能合约是可以自动化执行一些预先定义好的规则和条款的一段计算机程序代码,它大大提高了经济活动与契约的自动化程度。

3. 可信任

存储在区块链上的交易记录和其他数据是不可篡改并且可溯源的,所以能够很好地解决各方不信任的问题,无需第三方可信中介。

1.1.3 区块链相关概念

区块链以密码学的方式维护一份不可篡改和不可伪造的分布式账本,并通过基于协商一致的

规范和协议（共识机制）解决了去中心化的记账系统的一致性问题^[10]，其相关概念主要包括以下三个。

- 交易（Transaction）：区块链上每一次导致区块状态变化的操作都称为交易，每一次交易对应唯一的交易哈希值，一段时间后便会对交易进行打包。
- 区块（Block）：打包记录一段时间内发生的交易和状态结果，是对当前账本的一次共识。每个区块以一个相对平稳的时间间隔加入到链上，在企业级区块链平台中，共识时间可以动态设置。
- 链（Chain）：区块按照时间顺序串联起来，通过每个区块记录上一个区块的哈希值关联，是整个状态改变的日志记录。

图1.1展示的区块链主要结构可以帮助大家理解这些概念。

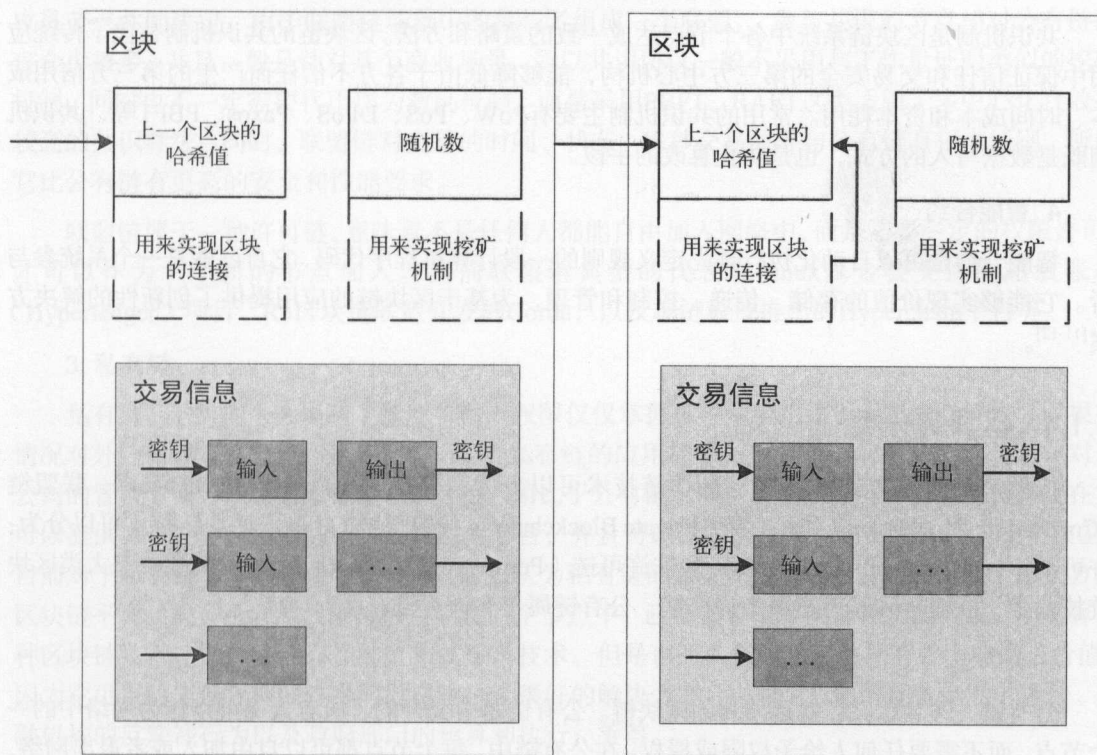


图1.1 区块链主要结构

区块链技术体系不是通过一个权威的中心化机构来保证交易的可信和安全，而是通过加密和分布式共识机制来解决信任和安全问题，其主要技术创新有以下4点。

1. 分布式账本

交易是由分布式系统中的多个节点共同记录的。每一个节点都记录完整的交易记录，因此它

们都可以参与监督交易合法性并验证交易的有效性。不同于传统的中心化技术方案，区块链中没有任何一个节点有权限单独记录交易，从而避免了因单一记账人或节点被控制而造假的可能性。另一方面，由于全网节点参与记录，理论上讲，除非所有的节点都被破坏，否则交易记录就不会丢失，从而保证了数据的安全性。

2. 加密技术和授权技术

区块链技术很好地集成了当前对称加密、非对称加密和哈希算法的许多优点，并使用了数字签名技术来保证交易的安全性，其中最具代表性的是使用椭圆曲线加密算法生成用户的公私钥对和使用椭圆曲线数字签名算法来保证交易安全。打包在区块上的交易信息对于参与共识的所有节点是公开的，但是账户的身份信息是经过严格加密的。

3. 共识机制

共识机制是区块链系统中各个节点达成一致的策略和方法。区块链的共识机制替代了传统应用中保证信任和交易安全的第三方中心机构，能够降低由于各方不信任而产生的第三方信用成本、时间成本和资本耗用。常用的共识机制主要有PoW、PoS、DPoS、Paxos、PBFT等，共识机制既是数据写入的方式，也是防止篡改的手段。

4. 智能合约

智能合约是可以自动化执行预先定义规则的一段计算机程序代码，它自己就是一个系统参与者。它能够实现价值的存储、传递、控制和管理，为基于区块链的应用提供了创新性的解决方案^[11-12]。

1.1.4 区块链分类

按照节点参与方式的不同，区块链技术可以分为：公有链（Public Blockchain）、联盟链（Consortium Blockchain）和私有链（Private Blockchain）。按照权限的不同，区块链技术可以分为：许可链（Permissioned Blockchain）和非许可链（Permissionless Blockchain）。前述的三大类区块链技术中，联盟链和私有链属于许可链，公有链属于非许可链。

1. 公有链

公有链，顾名思义，就是公开的区块链。公有链是全公开的，所有人都可以作为网络中的一个节点，而不需要任何人给予权限或授权。在公有链中，每个节点都可以自由加入或者退出网络，参与链上数据的读写、执行交易，还可以参与网络中共识达成的过程，即决定哪个区块可以添加到主链上并记录当前的网络状态^[13]。公有链是完全意义上的去中心化区块链，它借助密码学中的加密算法保证链上交易的安全。在采取共识算法达成共识时，公有链主要采取工作量证明（PoW, Proof of Work）机制或权益证明（PoS, Proof of Stake）机制等共识算法，将经济奖励和加密数字验证结合起来，来达到去中心化和全网达成共识的目的。在这些算法共识形成的过程中，每个节点都可以为共识过程做出贡献，也是我们俗称的“挖矿”，来获取与贡献成正比的经济奖励，也

就是系统中发行的数字代币。

公有链通常也被称为公共链,它属于一种非许可链,不需要许可就可以自由参加退出。当前最典型的代表应用有比特币、以太坊(Ethereum)等。因其完全去中心化和面向大众的特性,公有链通常适用于“虚拟加密货币”和面向大众的一些金融服务以及电子商务等。

2. 联盟链

联盟链不是完全去中心化的,而是一种多中心化或者部分去中心化的区块链。在区块链系统运行时,它的共识过程可能会受某些指定节点的控制。例如,在一个有15个金融机构接入的区块链系统中,每个机构都作为链上的一个节点,每确认一笔交易,都需要至少对10个节点进行确认(2/3确认),这笔交易或者这个区块才能被认可。联盟链账本上的数据与公有链的完全公开是不同的,只有联盟成员节点才可以访问,并且链上的读写权限、参与记账规则等操作也需要由联盟成员节点共同决定。由于联盟链场景中的参与者组成一个联盟,参与共识的节点相对公有链而言会少很多,并且一般是针对某个商业场景,所以共识协议一般不采用与工作量证明类似的挖矿机制,同时也不一定需要代币作为激励机制,而是采用PBFT、RAFT这类适用于多中心化且效率较高的共识算法。同时,联盟链对交易的时间、状态、每秒交易数等与公有链有很大区别,所以它比公有链有更高的安全和性能要求。

联盟链属于一种许可链,意味着不是任何人都能自由加入网络中,而是需要一定的权限许可,才可以作为一个新的节点加入。当前联盟链典型的代表有Linux基金会支持的超级账本(Hyperledger)项目、R3区块链联盟开发的Corda,以及趣链科技推出的Hyperchain平台等。

3. 私有链

私有链,是指整个区块链上的所有写入权限仅仅掌握在一个组织手里,而读取权限可以根据情况对外开放或者任意进行限制^[14]。所以,私有链的应用场景一般是单一的企业内部总公司对分公司的管理方面,如数据库管理和审计等。相比于公有链和联盟链,私有链的价值主要体现在它可以提供一个安全、可追溯、不可篡改的平台,并且可以同时防止来自内部和外部的安全攻击。目前对于私有链确实存在着一些争议,有人认为私有链的意义不大,因为它需要依赖于第三方的区块链平台机构,所有的权限都被控制在一个节点中,已经违背了区块链技术的初衷,不能算一种区块链技术,而是已经存在的分布式账本技术。但是也有人认为私有链拥有很大的潜在价值,因为它可以给当前存在的许多问题提供一个很好的解决方案,比如企业内部规章制度的遵守、金融机构的反洗钱行为以及政府部门的预算和执行,等等。

与联盟链一样,私有链也属于一种许可链,不过它的许可权掌握在单一节点中,在有些场景中,私有链还被称为专有链。当下私有链的应用不是很多,开创者都在努力探索之中。当前已经存在的应用主要有英国币科学公司(Coin Sciences Ltd.)推出的多链(Multichain)平台,这个平台的宗旨是希望能帮助各企业快速地部署私链环境,提供良好的隐私保护和权限控制。

自诞生至今,区块链技术经历了三次大的技术演进,其典型代表平台为2009年的比特币、2013年的以太坊和2015年的Fabric和Hyperchain,其组织形态从资源消耗严重、交易性能低下、缺乏

灵活控制机制的公有区块链，向高效共识、智能可编程、可保护隐私的联盟区块链转变。当前，Hyperchian平台的TPS（每秒事务处理量）已达到千甚至万量级，可以满足大部分商业场景的需要。将来，随着技术的进一步发展，基于联盟链的区块链商业应用将成为区块链应用的主要形态。

1.2 区块链发展历程

比特币所实现的基于零信任基础、真正去中心化的分布式系统，其实是解决一个30多年前由Leslie Lamport等人提出的拜占庭将军问题。区块链技术从诞生至今，其发展历程大致可以分为4个阶段：技术起源、区块链1.0、区块链2.0和区块链3.0，如图1.2所示。

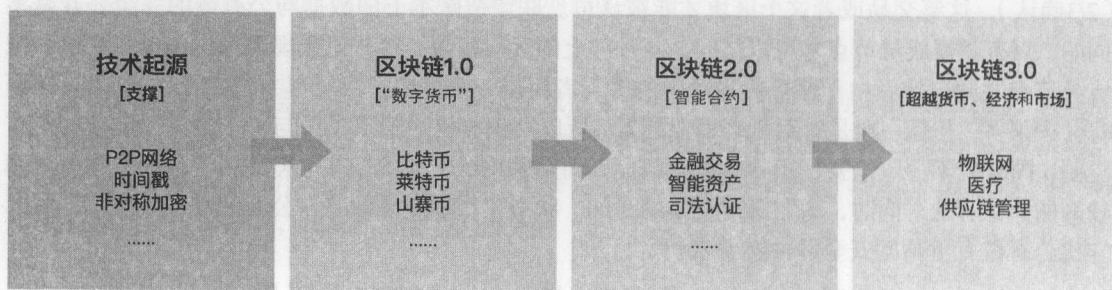


图1.2 区块链发展历程

1.2.1 技术起源

区块链技术源于中本聪创造的比特币。比特币是中本聪站在巨人的肩膀上，基于前人的各种相关技术和算法，结合自己独特的创造性思维而设计出来的。下面简要介绍区块链相关基础技术的发展历史^[15]。

1982年，Leslie Lamport等人提出拜占庭将军问题（Byzantine Generals Problem）^[16]，这是一个非常著名的、分布式计算领域的问题，旨在设法建立具有容错性的分布式系统，即在一个存在故障节点和错误信息的分布式系统中保证正常节点达到共识，保持信息传递的一致性。

1985年，Neal Koblitz和Victor Miller两人提出椭圆曲线密码学（Elliptic Curve Cryptography, ECC）^[17-18]，第一次将椭圆曲线用于密码学中，建立公开金钥加密演算法。相较于之前的RSA演算法，采用ECC的好处在于可用较短的金钥达到与RSA相同的安全强度。

1990年，David Chaum根据之前提出的密码学网络支付系统理念，实现了一个不可追踪密码学网络支付系统，称为eCash^[19]。不过，这是一个中心化的系统，但区块链技术在隐私安全上借鉴了其很多设计。

1990年，Leslie Lamport针对自己在1982年提出的拜占庭将军问题，给出了一个解决方案——Paxos算法^[20]，Paxos共识算法能在分布式系统中达成高容错性的全网一致性。

1991年, Stuart Haber与W. Scott Stornetta提出了时间戳技术来确保电子文件安全^[21], 中本聪在比特币中也采用了这一技术, 对账本中的交易进行追本溯源。

1992年, Scott Vanstone等人基于ECC提出了性能更好的椭圆曲线数字签名算法(Elliptic Curve Digital Signature Algorithm, ECDSA)^[22]。

1997年, Adam Back发明了Hashcash, 一种工作量证明算法^[23], 此演算法仰赖成本函数的不可逆特性, 具有容易被验证但很难被破解的特性, 最早被应用于阻挡垃圾邮件。其算法设计理念被中本聪改进之后, Hashcash成为比特币区块链节点达成共识的核心技术之一, 是比特币的基石。

1998年, Wei Dai发表了匿名的分布式电子现金系统B-money^[24], 引入了工作量证明机制, 强调点对点交易和不可篡改特性。不过在B-money中, 并未采用Adam Back提出的Hashcash演算法。Wei Dai的许多设计也被比特币区块链所采用。

2005年, Hal Finney提出可重复使用的工作量证明机制(Reusable Proofs of Work, RPOW)^[25], 结合B-money与Adam Back提出的Hashcash演算法来创造密码学“货币”。

2008年, 中本聪在一个隐秘的密码学讨论组发表了一篇关于比特币的论文, 发明了比特币^[5]。

从上述技术发展历史来看, 区块链技术并不是一蹴而就的, 而是一定背景和技术发展下的必然产物。关于区块链的核心技术, 后续章节会进行系统性的详细介绍。

1.2.2 区块链 1.0——“数字货币”

在区块链1.0阶段, 区块链技术的应用范围主要集中在“数字货币”领域。在2009年比特币上线之后, 由于比特币区块链解决了“双花问题”和“拜占庭将军问题”, 真正扫清了“数字货币”流通的主要障碍, 因而获得了极大的追捧, 狗狗币、莱特币之类的“山寨”“数字货币”也开始大量涌现。这些“数字货币”在技术上与比特币十分类似, 其架构一般都可分为三层: 区块链层、协议层和货币层。区块链层作为这些“数字货币”系统的底层技术, 是最核心部分, 系统的共识过程、消息传递等核心功能都是通过区块链达成的。协议层则主要为系统提供一些软件服务、制定规则, 等等。最后的货币层则主要是作为价值表示, 用来在用户之间传递价值, 相当于一种货币单位。

在区块链1.0阶段, 基于区块链技术构建了很多去中心化数字支付系统, 很好地解决了货币和支付手段的去中心化问题, 对传统的金融体系有着一定的冲击。

1.2.3 区块链 2.0——智能合约

在比特币和其他山寨币的资源消耗严重、无法处理复杂逻辑等弊端逐渐暴露后, 业界逐渐将关注点转移到了比特币的底层支撑技术区块链上, 产生了运行在区块链上的模块化、可重用、自动执行脚本, 即智能合约。这大大拓展了区块链的应用范围, 区块链由此进入2.0阶段。业界慢慢地认识到区块链技术潜藏的巨大价值。区块链技术开始脱离“数字货币”领域的创新, 其应用

范围延伸到金融交易、证券清算结算、身份认证等商业领域。涌现了很多新的应用场景，如金融交易、智能资产、档案登记、司法认证，等等。

以太坊是这一阶段的代表性平台，它是一个区块链基础开发平台，提供了图灵完备的智能合约系统。通过以太坊，用户可以自己编写智能合约，构建去中心化的DAPP。基于以太坊智能合约图灵完备的性质，开发者可以编程任何去中心化应用，例如投票、域名、金融交易、众筹、知识产权、智能财产，等等。目前在以太坊平台运行着很多去中心化应用，按照其白皮书说明，它们可以分为三种应用。第一种是金融应用，包括“数字货币”、金融衍生品、对冲合约、储蓄钱包、遗嘱这些涉及金融交易和价值传递的应用。第二种是半金融应用，它们涉及金钱的参与，但有很大一部分是非金钱的方面。第三种则是非金融应用，如在线投票和去中心化自治组织这类不涉及金钱的应用。

在区块链2.0阶段，以智能合约为主导，越来越多的金融机构、初创公司 and 研究团体加入了区块链技术的探索队列，推动了区块链技术的迅猛发展。

1.2.4 区块链 3.0——超越货币、经济和市场

随着区块链技术的不断发展，区块链技术的低成本信用创造、分布式结构和公开透明等特性的价值逐渐受到全社会的关注，在物联网、医疗、供应链管理、社会公益等各行各业中不断有新应用出现。区块链技术的发展进入到了区块链3.0阶段^[11]。在这一阶段，区块链的潜在作用并不仅仅体现在货币、经济和市场方面，更延伸到了政治、人道主义、社交和科学领域，区块链技术方面的能力已经可以让特殊的团体来处理现实中的问题。而随着区块链的继续发展，我们可以大胆构想，区块链技术或许将广泛而深刻地改变人们的生活方式，并重构整个社会，重铸信用价值。或许将来当区块链技术发展到一定程度时，整个社会中的每一个人都可作为一个节点，连接到一个全球性的去中心化网络中，整个社会进入区块链时代，然后通过区块链技术来分配社会资源，或许区块链将成为一个促进社会经济理想发展的理想框架。

1.3 区块链关键技术

通过对区块链基础知识和发展历程的介绍，相信读者已经对区块链有了一个较为直观的认识，本节将更进一步，深入介绍区块链的系统架构和关键技术。

1.3.1 基础模型

图1.3所示是区块链的基本架构，该图的绘制参考了《区块链技术发展现状与展望》^[2]和工信部《中国区块链技术和应用发展白皮书(2016)》^[9]中的区块链架构图。区块链基本架构可以分为数据层、网络层、共识层、激励层、合约层和应用层：

- 数据层封装了区块链的链式结构、区块数据以及非对称加密等区块链核心技术；

- 网络层提供点对点的数据通信传播以及验证机制；
- 共识层主要是网络节点间达成共识的各种共识算法；
- 激励层将经济因素引入到区块链技术体系之中，主要包括经济因素的发行机制和分配机制；
- 合约层展示了区块链系统的可编程性，封装了各类脚本、智能合约和算法；
- 应用层则封装了区块链技术的应用场景和案例。

在该架构中，基于时间戳的链式结构、分布式节点间的共识机制和可编程的智能合约是区块链技术最具代表性的创新点。一般可以在合约层编写智能合约或者进行脚本编程，来构建基于区块链的去中心化应用。下面将对本架构中每一层所涉及的技术展开具体介绍。

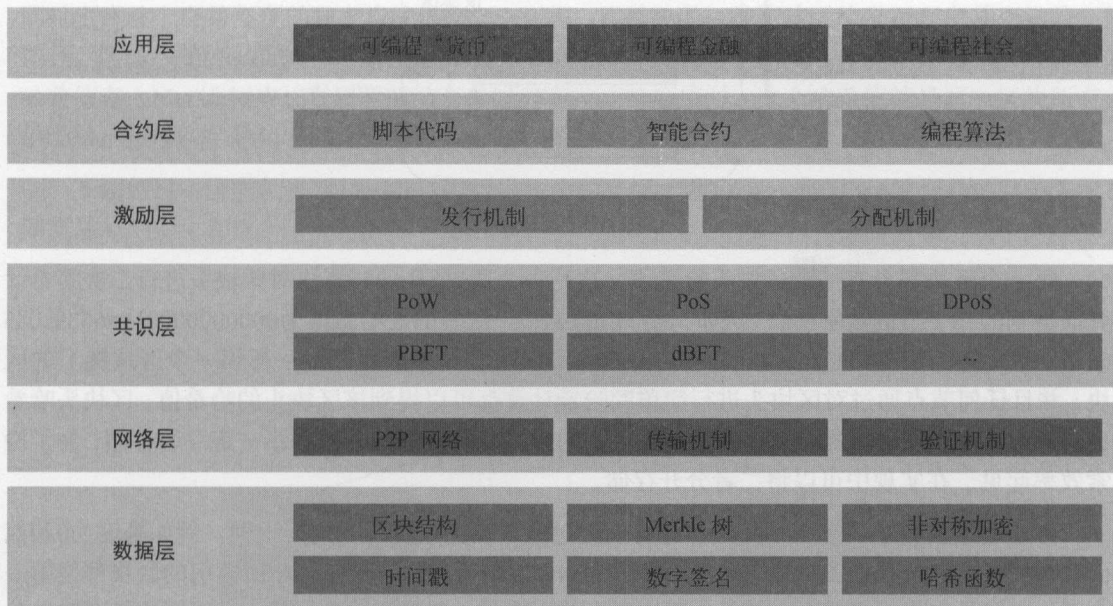


图1.3 区块链基本架构

1.3.2 数据层

数据层是区块链的核心部分，区块链本质上是一种数据库技术和分布式共享账本，是由包含交易信息的区块从后向前有序连接起来的一种数据结构。该层涉及的技术主要包括：区块结构、Merkle树、非对称加密、时间戳、数字签名和哈希函数。时间戳和哈希函数相对比较简单，这里重点介绍一下区块结构、Merkle树、非对称加密和数字签名。

1. 区块结构

每个区块一般都由区块头和区块体两部分组成。如图1.4所示，区块头部分包含了父区块哈希值、时间戳、Merkle根等信息，而区块体部分则包含着此区块中所有的交易信息。除此之外，每一个区块还对应着两个值来识别区块：区块头哈希值和区块高度。

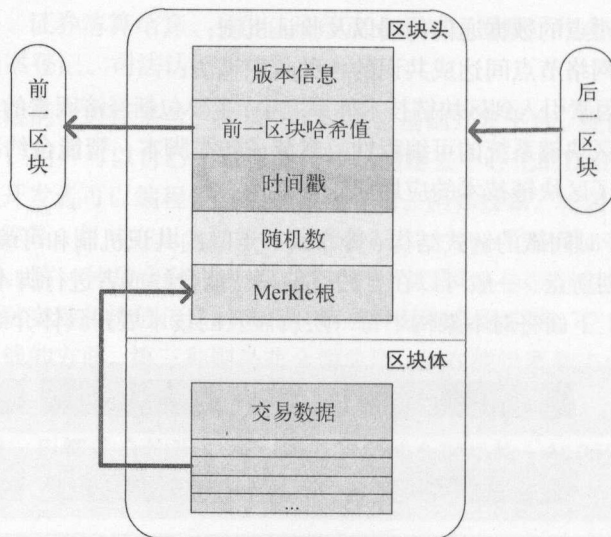


图1.4 区块结构

每一个区块都会有一个区块头哈希值，这是一个通过SHA256算法对区块头进行二次哈希计算而得到的32字节的数字指纹。例如，比特币的第一个区块的头哈希值为000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f。区块头哈希值可以唯一标识一个区块链上的区块，并且任何节点通过对区块头进行简单的哈希计算都可以得到该区块头的哈希值。区块头哈希也包含在区块的整体数据结构中，但是区块头的数据和区块体的数据并不一定一起存储，为了检索效率起见，在实现中可以将二者分开存储。

除了通过头哈希值来识别区块，还可以通过区块高度来对区块进行识别。例如高度为0和前面000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f所索引的区块都是第一个区块。但是与头哈希值不同的是，区块高度并不能唯一地标识一个区块。由于区块链存在着分叉情况，所以可能存在2个或以上区块的区块高度是一样的。

谈完了头哈希值和区块高度，下面介绍区块头的构造。以比特币为例，区块头是80字节，其详细结构如表1.1所示^[7]。

表1.1 区块头详细结构

字 段	大小 (字节)	描 述
版本	4	版本号，用于跟踪软件/协议的更新
前一区块哈希值	32	引用区块链中前一区块的哈希值
Merkle根	32	该区块中交易的Merkle树根的哈希值
时间戳	4	该区块产生的近似时间（精确到秒的Unix时间戳）
随机数	4	用于工作量证明算法的计数器

区块头由三组元数据组成，一组是引用父区块的哈希值数据，用于同前一区块进行相连。第二组即难度值、时间戳和随机数，这些都与挖矿竞争相关。第三组是Merkle根，是区块体中Merkle树的根节点。

2. Merkle树

前面介绍了区块头哈希值、区块高度和区块头的结构，接着来看看区块体。区块体存储着交易信息，在区块中它们是以一棵Merkle树的数据结构进行存储的，而Merkle树是一种用来有效地总结区块中所有交易的数据结构。Merkle树是一棵哈希二叉树，树的每个叶子节点都是一笔交易的哈希值。同样以比特币为例，在比特币网络中，Merkle树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹即Merkle树根，且提供了一种校验区块是否存在某交易的高效途径。生成一棵Merkle树需要递归地对每两个哈希节点进行哈希得到一个新的哈希值，并将新的哈希值存入Merkle树中，直到两两结合最终只有一个哈希值时，这个哈希值就是这一区块所有交易的Merkle根，存储到上面介绍的区块头结构中。

下面通过一个实例来对Merkle树进行进一步的介绍。图1.5是一棵只有4笔交易的Merkle树，即交易A、B、C和D。

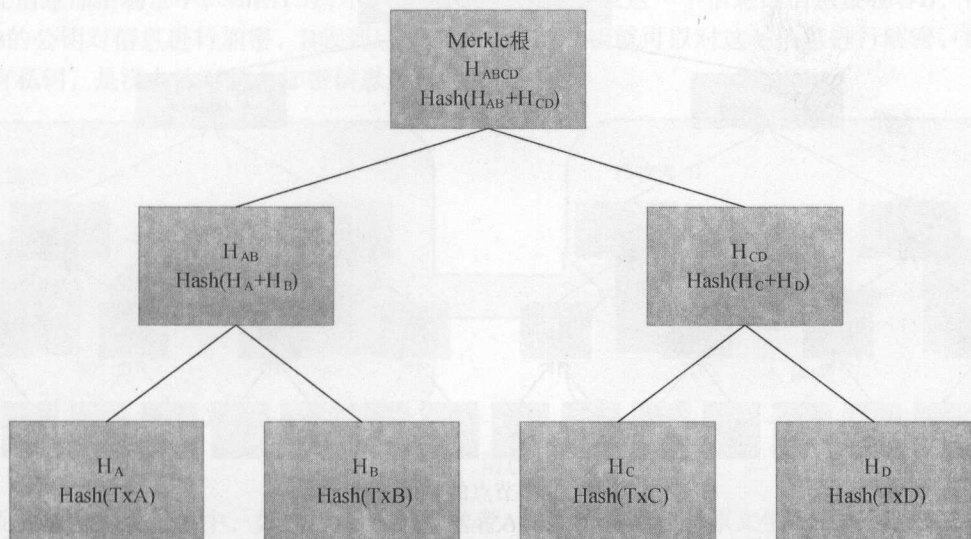


图1.5 Merkle树

第一步，需要使用两次SHA256算法对每笔交易数据进行哈希运算，得到每笔交易的哈希值，这里可以得到 H_A 、 H_B 、 H_C 、 H_D 这4个哈希值，也就是这棵Merkle树的叶子节点。例如，

$$H_A = \text{SHA256}(\text{SHA256}(\text{交易A}))$$

第二步，对两个叶子节点 H_A 、 H_B 的哈希值同样使用两次SHA256进行组合哈希运算，将会得

到一个新的哈希值 H_{AB} ，对 H_C 、 H_D 进行同样的操作将获得另一个哈希值 H_{CD} 。例如，

$$H_{AB} = \text{SHA256}(\text{SHA256}(H_A + H_B))$$

第三步，对现有的两个哈希值 H_{AB} 、 H_{CD} 进行第二步中的组合运算，最后将得到一个新的哈希值 H_{ABCD} ，此时我们已经没有了其他同高度节点，所以最后的 H_{ABCD} 就是这一棵Merkle树的Merkle根。之后将这个节点的32字节哈希值写入到区块头部Merkle根字段中。Merkle树的整个形成过程结束。

$$H_{ABCD} = \text{SHA256}(\text{SHA256}(H_{AB} + H_{CD}))$$

因为Merkle树是一棵二叉树，所以它需要偶数个叶子节点，也就是偶数笔交易。但是在很多情况下，某个区块的交易数目会出现奇数笔。对于这种情况，Merkle树的解决方案是将最后一笔交易进行一次复制，以此构造成偶数个叶子节点，这种偶数个叶子节点的二叉树也称为平衡树。

图1.6展示的是一棵更大的Merkle树，由16个交易构成。通过图示，可以发现，不管一个区块中有一笔交易还是十万笔交易，最终都能归纳成一个32字节的哈希值作为Merkle树的根节点。

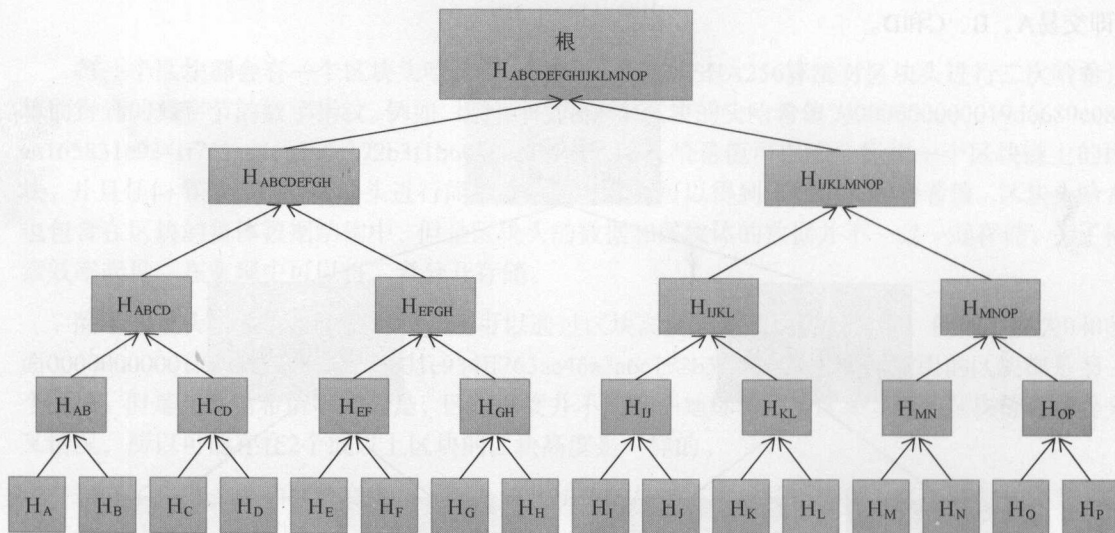


图1.6 多节点的Merkle树

当需要证明交易列表中的某笔交易存在时，一个节点只需计算 $\log_2 N$ 个32字节的哈希值，就可以形成一条从Merkle树根到特定交易的路径，Merkle树的效率如表1.2所示^[7]。

表1.2 Merkle树效率

交易数量（笔）	区块的近似大小（千字节）	路径大小（哈希数量）	路径大小（字节）
16	4	4	128
512	128	9	288

(续)

交易数量 (笔)	区块的近似大小 (千字节)	路径大小 (哈希数量)	路径大小 (字节)
2048	512	11	352
65535	16384	16	512

3. 非对称加密与数字签名

非对称加密是区块链技术中用于安全性需求和所有权认证时采用的加密技术,常见的非对称加密算法有RSA、Elgamal、背包算法、Rabin、D-H、ECC (椭圆曲线加密算法) 和ECDSA (椭圆曲线数字签名算法),等等^[9-26]。与对称加密算法不同的是,非对称加密算法需要两个密钥:公开密钥 (public key) 和私有密钥 (private key)。基于非对称加密算法可使通信双方在不安全的媒体上交换信息,安全地达成信息的一致。公开密钥是对外公开的,而私有密钥是保密的,其他人不能通过公钥推算出对应的私钥。每一个公开密钥都有其相对应的私有密钥,如果我们使用公开密钥对信息进行了加密,那么则必须有对应的私有密钥才能对加密后的信息进行解密;而如果是用私有密钥加密信息,则只有对应的公开密钥才可以进行解密。在区块链中,非对称加密主要用于信息加密、数字签名等场景。

在信息加密场景中,如图1.7所示,信息发送者A需要发送一个信息给信息接收者B,需要先使用B的公钥对信息进行加密,B收到后,使用自己的私钥就可以对这一信息进行解密,而其他

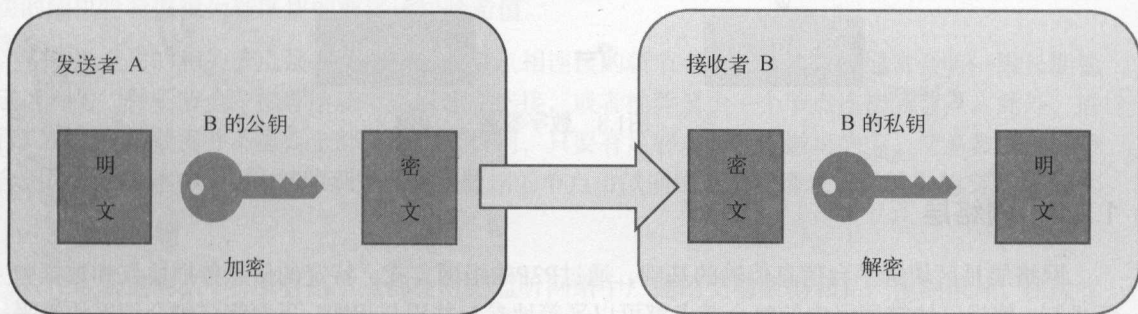


图1.7 信息加密

而在数字签名场景中,如图1.8所示,发送者A先用哈希函数对原文生成一个摘要 (Digest),然后使用私钥对摘要进行加密,生成数字签名 (Signature),之后将数字签名与原文一起发送给接收者B;B收到信息后使用A的公钥对数字签名进行解密得到摘要,由此确保信息是A发出的,然后再对收到的原文使用哈希函数产生摘要,并与解密得到的摘要进行对比,如果相同,则说明收到的信息在传输过程中没有被修改过。

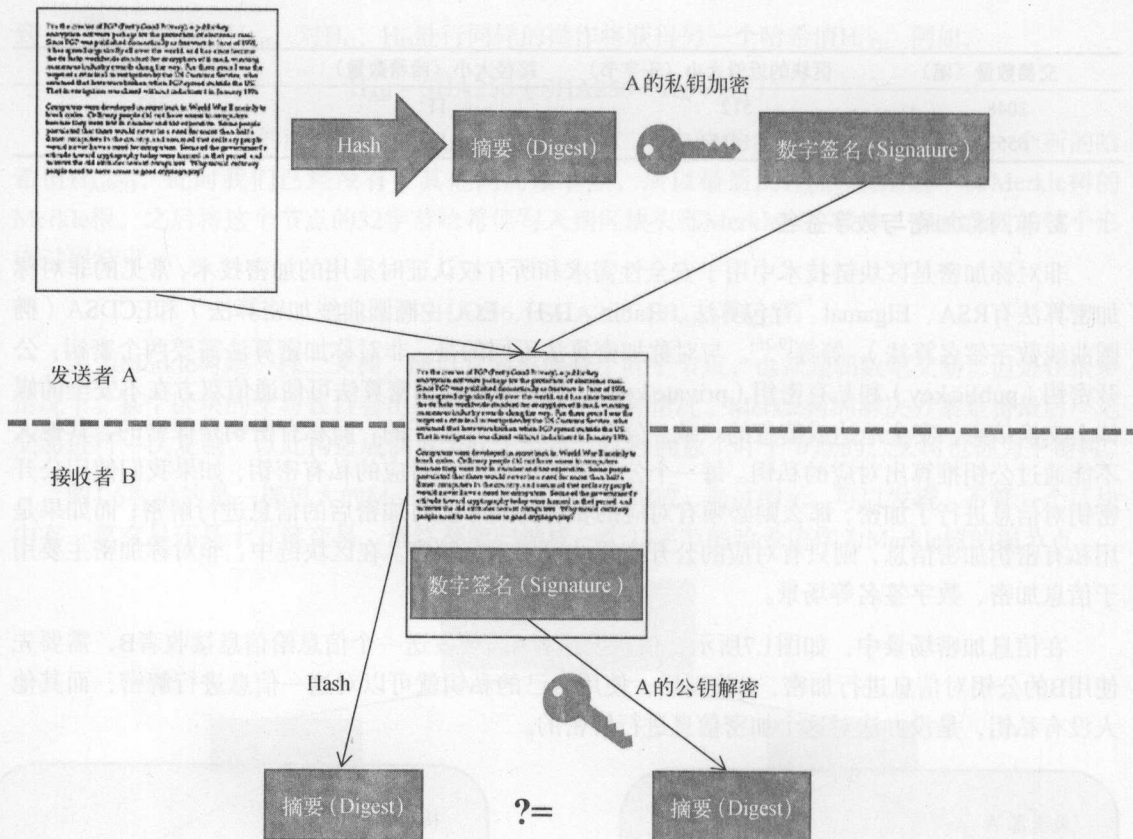


图1.8 数字签名

1.3.3 网络层

网络层是区块链平台信息传输的基础，通过P2P的组网方式、特定的信息传播协议和数据验证机制，使得区块链网络中的每个节点都可以平等地参与共识与记账。下面将详细介绍区块链平台网络层中的P2P网络架构、信息传输机制和数据验证机制。

1. P2P网络架构

区块链网络架构一般采用的是基于互联网的P2P (peer-to-peer) 架构，在P2P网络中，每台计算机每个节点都是对等的，它们共同为全网提供服务。而且，没有任何中心化的服务端，每台主机都可以作为服务端响应请求，也可以作为客户端使用其他节点所提供的服务。P2P通信不需要从其他实体或CA获取地址验证，因此有效地消除了篡改的可能性和第三方欺骗^[27]。所以P2P网络是去中心化和开放的，这也正符合区块链技术的理念。

在区块链网络中，所有的节点地位均等且以扁平式拓扑结构相互连通和交互，每个节点都需

要承担网络路由、验证区块数据、传播区块数据等功能。在比特币网络中,存在着两类节点,一类是全节点,它保存着区块链上所有的完整数据信息,并需要实时地参与区块链数据的校验和记录来更新区块链主链。另一类是轻节点,它只保存着区块链中的部分信息,通过简易支付验证(SPV)方式向其他相邻的节点请求数据以便完成数据的验证^[2]。

2. 传输机制

在新的区块数据生成后,生成该数据的节点会将其广播到全网的其他节点以供验证。目前的区块链底层平台一般都会根据自身的实际应用需求,在比特币传输机制的基础上重新设计或者改进出新的传输机制,如以太坊区块链集成了所谓的“幽灵协议”,以解决因区块数据确认速度快而导致的高区块作废率和随之而来的安全性风险^[28]。这里我们以中本聪设计的比特币系统为例,列出其传输协议的步骤如下^[5]:

- (1) 比特币交易节点将新生成的交易数据向全网所有节点进行广播;
- (2) 每个节点都将收集到的交易数据存储到一个区块中;
- (3) 每个节点基于自身算力在区块中找到一个具有足够难度的工作量证明;
- (4) 当节点找到区块的工作量证明后,就向全网所有节点广播此区块;
- (5) 只有包含在区块中的所有交易都有效且之前未存在过,其他节点才认同该区块的有效性;
- (6) 其他节点接收该数据区块,并在该区块的末尾制造新的区块以延长链,而将被接收的区块的随机哈希值视为新区块的前序区块哈希值。

如果交易的相关节点是一个未与其他节点相连接的新节点,比特币系统通常会将一组长期稳定运行的“种子节点”推荐给新节点以建立连接,或者推荐至少一个节点连接新节点。此外,进行广播的交易数据并不需要全部节点都接收到,只要有足够多的节点做出响应,交易数据便可整合到区块链账本中。而未接收到完整交易数据的节点可以向临近节点请求下载缺失的交易数据^[7]。

3. 验证机制

在区块链网络中,所有的节点都会时刻监听网络中广播的交易数据和新产生的区块。在接收到相邻节点发来的数据后,会首先验证该数据的有效性,若数据有效则按接收顺序为新数据建立存储池来暂存这些数据,并且继续向临近节点转发;若数据无效则立即废弃该数据,从而保证无效数据不会在区块链网络中继续传播。验证有效性的方法是根据预定义好的标准,从数据结构、语法规范性、输入输出和数字签名等各方面进行校验。对于新区块的校验同理,某节点产生出新区块后,其他节点按照预定义的标准对新区块的工作量证明、时间戳等方面进行校验,若确认有效,则将该区块链接到主区块链上,并开始争取下一个区块的记账权。

1.3.4 共识层

Leslie Lamport于1982年提出著名的拜占庭将军问题,引发了无数研究者探索解决方案。如何

在分布式系统中高效地达成共识是分布式计算领域的一个重要研究课题。区块链的共识层的作用就是在不同的应用场景下通过使用不同的共识算法,在决策权高度分散的去中心化系统中使得各个节点高效地达成共识。

最初,比特币区块链选用了一种依赖节点算力的工作量证明共识(Proof of Work, PoW)机制来保证比特币网络分布式记账的一致性。之后随着区块链技术的不断演进和改进,研究者陆续提出了一些不过度依赖算力而能达到全网一致的算法,比如权益证明共识(Proof of Stake, PoS)机制、授权股份证明共识(Delegated Proof of Stake, DPoS)机制、实用拜占庭容错(Practical Byzantine Fault Tolerance, PBFT)算法,等等。下面我们对这几种共识算法进行简单介绍^[29]。

1. PoW (工作量证明机制)

PoW机制诞生于1997年Adam Back设计的Hashcash系统,它最初被创造出来用于预防邮件系统中漫天遍地的垃圾邮件^[11]。2009年,中本聪将PoW机制运用于比特币区块链网络中,作为达成全网一致性的共识机制。从严格意义上讲,比特币中所采用的是一种可重复使用的Hashcash工作证明,使得生成工作证明量可以是一个概率意义上的随机过程^[29]。在该机制中,网络上的每一个节点都在使用SHA256哈希算法运算一个不断变化的区块头的哈希值。共识要求算出的值必须等于或者小于某个给定的值^[30]。在分布式网络中,所有的参与者都需要使用不同的随机数来持续计算该哈希值,直到达到目标为止。当一个节点得出了确切的值,其他所有的节点必须相互确认该值的正确性。之后,新区块中的交易将被验证以防欺诈。然后,用于计算的交易信息的集合会被确认为认证结果,用区块链中的新区块表示。在比特币中,运算哈希值的节点被称作“矿工”,而PoW的过程被称为“挖矿”。由于认证的计算是一个耗时的过程,所以也提出了相应的激励机制(例如向矿工授予一小部分比特币)。总的来说,工作量证明就是对于工作量的证明,每个区块加入到链上,必须得到网络参与者的同意验证,矿工对它完成了相对应的工作量。PoW的优点是完全的去中心化和分布式账簿。缺点也很明显,即消耗资源:挖矿行为造成了大量的资源浪费,同时PoW达成共识的周期也比较长,比特币网络会自动调整目标值来确保区块生成过程大约需要10分钟,因此它不是很适合商业运用。

2. PoS (股权证明机制)

PoS的想法源于尼克·萨博(Nick Szabo),是PoW的一种节能替代选择,它不需要用户在不受限制的空间中找到一个随机数,而是要求人们证明货币数量的所有权,因为其相信拥有货币数量多的人攻击网络的可能性更低。由于基于账户余额的选择是非常不公平的,因为单一最富有的人势必在网络中占主导地位,所以提出了许多解决方案,结合股权来决定谁来创建下一个块。其中,Blackcoin使用随机选择来预测下一个创建者,而Peercoin则倾向于基于币龄来选择。Peercoin首次开创性地实现了真正的股权证明,它采用工作量证明机制发行新币,采用股权证明机制维护网络安全,这也是“虚拟货币”历史上的一次创举。同比特币网络要求证明人执行一定量的工作不同,该机制只需要证明人提供一定数量“数字货币”的所有权即可。在股权证明机制中,每当创建一个区块时,矿工需要创建一个称为“币权”的交易,这个交易会按照一定的比例预先将一

些币发给矿工。然后股权证明机制根据每个节点持有代币的比例和时间,依据算法等比例地降低节点的挖矿难度,以加快节点寻找随机数的速度,缩短达成共识所需的时间^[31]。与PoW相比,PoS可以节省更多的能源,更有效率。但是,由于挖矿成本接近于零,因此可能会遭受攻击。且PoS在本质上仍然需要网络中的节点进行挖矿运算,所以它同样难以应用于商业领域。

3. DPoS (股份授权证明机制)

DPoS由比特股(Bitshares)项目组发明^[32]。股权拥有者选举他们的代表来进行区块的生成和验证。DPoS类似于现代企业董事会制度,比特股系统将代币持有者称为股东,由股东投票选出101名代表,然后由这些代表负责生成和验证区块。持币者若想成为一名代表,需先用自己的公钥去区块链注册,获得一个长度为32位的特有身份标识符,股东可以对这个标识符以交易的形式进行投票,得票数前101位被选为代表。代表们轮流产生区块,收益(交易手续费)平分。如果有的代表不老实生产区块,很容易被其他代表和股东发现,他将立即被踢出“董事会”,空缺位置由票数排名102的代表自动填补。DPoS的优点在于大幅减少了参与区块验证和记账的节点数量,从而缩短了共识验证所需要的时间,大幅提高了交易效率。从某种角度来说,DPoS可以理解得多中心系统,兼具去中心化和中心化优势。

4. PBFT (实用拜占庭容错算法)

这个算法最初出现在MIT的Miguel和Barbara Liskov的学术论文中^[33],初衷是为一个低延迟存储系统所设计,降低算法的复杂度,该算法可以应用于吞吐量不大但需要处理大量事件的数字资产平台。它允许每个节点发布公钥,任何通过节点的消息都由节点签名,以验证其格式。验证过程分为三个阶段:预备、准备、落实。如果已经收到超过1/3不同节点的批准,服务操作将是有效的。使用PBFT,区块链网络 N 个节点中可以包含 f 个拜占庭恶意节点,其中 $f=(N-1)/3$ 。换句话说,PBFT确保至少 $2f+1$ 个节点在将信息添加到分布式共享账簿之前达到共识。目前,HyperLedger联盟、中国ChinaLedger联盟等诸多区块链联盟都在研究和验证这个算法的实际部署和应用。

1.3.5 激励层

激励层作为将经济因素引入区块链技术的一个层次,其存在的必要性取决于建立在区块链技术上的具体应用需求。这里以比特币系统为例,对其激励层进行介绍。

在比特币系统中,大量的节点算力资源通过共识过程得以汇聚,从而实现区块链账本的数据验证和记账工作,因而其本质上是一种共识节点间的任务众包过程^[2]。在去中心化系统中,共识节点本身是自利的,其参与数据验证和记账工作的根本目的是最大化自身收益。所以,必须设计合理的激励机制,使得共识节点最大化自身收益的个体行为与区块链系统的安全性和有效性相契合,从而使大规模的节点对区块链历史形成稳定的共识。

比特币采用PoW共识机制,在该共识中其经济激励由两部分组成:一是新发行的比特币;二是交易流通过程中的手续费。两者组合在一起,奖励给PoW共识过程中成功计算出符合要求的随

机数并生成新区块的节点。因此,只有当各节点达成共识,共同合作来构建和维护区块链历史记录及其系统的有效性,当作奖励的比特币才会有价值。

1. 发行机制

在比特币系统中,新区块产生发行比特币的数量是随着时间阶梯型递减的。从创世区块起,每个新区块将发行50个比特币奖励给该区块的记账者,此后每隔约4年(21万个区块),每个新区块发行的比特币数量减少一半,以此类推,一直到比特币的数量稳定在上限2100万为止^[7]。前文提到过,给记账者的另一部分奖励是比特币交易过程中产生的手续费,目前默认的手续费是1/10000个比特币。两部分费用会被封装在新区块的第一个交易(称为Coinbase交易)中。虽然现在每个新区块的总手续费与新发行的比特币相比要少得多,但随着时间推移,未来比特币的发行数量会越来越少,甚至停止发行,到那时手续费便会成为共识节点记账的主要动力。此外,手续费还可以起到保障安全性的作用,防止大量微额交易对比特币系统发起“粉尘攻击”。

2. 分配机制

随着比特币挖矿生态圈的成熟,“矿池”出现在人们的视野中。大量的小算力节点通过加入矿池而联合起来,相互合作汇集算力来提高获得记账权的概率,并共享生成新区块得到的新发行比特币和交易手续费奖励。据Bitcoinminning.com统计,目前已经存在13种不同的分配机制^[34]。现今主流矿池通常采用PPLNS(Pay Per Last N Shares)、PPS(Pay Per Share)和PROP(PRO Portionately)等机制。在矿池中,根据各个节点贡献的算力,按比例划分为不同的股份。PPLNS机制在产生新的区块后,各合作节点根据其在最后N个股份内贡献的实际股份比例来分配奖励;PPS则直接根据股份比例为各节点估算和支付一个固定的理论收益,采用此方式的矿池将会适度收取手续费来弥补其为各个节点承担的收益不确定性风险;PROP机制则根据节点贡献的股份按比例地分配奖励^[2]。

1.3.6 合约层

合约层封装了各类脚本、算法和智能合约,是区块链可编程性的体现。比特币本身就具有简单脚本的编写功能,而以太坊极大地强化了编程语言协议,理论上可以编写实现任何功能的应用^[28]。如果把比特币看成是全球账本的话,以太坊可以看作一台“全球计算机”,任何人都可以上传和执行任意的应用程序,并且程序的有效执行能得到保证。如果说数据、网络和共识三个层次作为区块链底层“虚拟机”,分别承担数据表示、数据传播和数据验证功能,合约层则是建立在区块链虚拟机之上的商业逻辑和算法,是实现区块链系统灵活编程和操作数据的基础。包括比特币在内的“数字加密货币”大多采用非图灵完备的简单脚本代码来编程控制交易过程,这也是智能合约的雏形。随着技术的发展,目前已经出现以太坊等图灵完备的可实现更为复杂和灵活的智能合约的脚本语言,使得区块链能够支持宏观金融和社会系统的诸多应用。

智能合约的概念可以追溯到1995年,是由学者尼克·萨博提出^[35]并进行如下定义的:“一个智能合约是一套以数字形式定义的承诺,包括合约参与方可以在上面执行这些承诺的协议。”其

设计初衷是希望通过将智能合约内置到物理实体来创造各种灵活可控的智能资产。但由于计算手段的落后和应用场景的缺失,智能合约在当时并未受到研究者的广泛关注^[2]。

区块链技术的出现对智能合约进行了新的定义并使其成为了可能。智能合约作为区块链技术的关键特性之一,是运行在区块链上的模块化、可重用、自动执行的脚本,能够实现数据处理、价值转移、资产管理等一系列功能。合约部署的时候被虚拟机编译成操作码存储在区块链上,对应地会有一个存储地址。当预定的条件发生时,就会发送一笔交易(transaction)到该合约地址,全网节点都会执行合约脚本编译生成的操作码,最后将执行结果写入区块链^[28,36-37]。作为一种嵌入式程序化合约,智能合约可以内置在任何区块链数据、交易或资产中,形成可由程序自行控制的系统、市场或资产。智能合约不仅为金融行业提供了创新性的解决方案,同时也能在社会系统中的信息、资产、合同、监管等事务管理中发挥重要作用。

基于区块链技术的智能合约不仅可以发挥智能合约在成本效率方面的优势,还可以避免恶意行为对合约正常执行的干扰。智能合约可以应用到任何一种数据驱动的业务逻辑中,以太坊首先看到了区块链和智能合约的契合,发布了白皮书《以太坊:下一代智能合约和去中心化应用平台》^[28],构建了内置有图灵完备编程语言的公有区块链,使得任何人都能够创建合约和去中心化应用。

智能合约与区块链的结合,丰富了区块链本身的价值内涵,其特性有以下3点^[38-39]:

- 用程序逻辑中的丰富合约规则表达能力实现了不信任方之间的公平交换,避免了恶意方中断协议等可能性;
- 最小化交易方之间的交互,避免了计划外的监控和跟踪的可能性;
- 丰富了交易与外界状态的交互,比如可信数据源提供的股票信息、天气预报等。

1.4 区块链产业现状

新技术的发展离不开市场和产业的推动,对于区块链技术的学习,仅仅了解其技术原理是不够的,还需知晓当前相关产业的发展情况。本节将从区块链技术的发展态势、政府对区块链技术的发展规划以及区块链产业生态图谱这3个维度,对区块链产业的发展现状进行分析介绍。

1.4.1 区块链发展态势

据统计,截止到2017年底,全球已有20多个国家开始投资发展区块链技术,并且有将近80%的银行表示会启动区块链项目,有90多个中央银行加入了区块链讨论,以及90多个公司加入了区块链联盟,与区块链技术相关的风险投资累计超过了20亿美元,并且产生了3000多个区块链相关专利^[40]。

在“数字货币”方面,截至2017年6月28日,共有928个“数字加密货币”,其中722个有市值统计,总市值超过了一千亿美元。其中,比特币市值423.36亿美元,占比39.89%;以太币市值301

亿美元,占比28.36%;瑞波币市值106亿美元,占比10%。前三大“数字加密货币”合计占比78.25%,第4~10名合计占比8.81%,第11~50名合计占比9.54%,其余672个币种仅占3.40%。这些数据表明区块链技术在“数字加密货币”领域充分展现了它的价值^[41]。

ICO (Initial Coin Offering) 是与“数字货币”密切相关的概念,是基于“数字货币”的项目初期资金筹措方式,早期参与ICO众筹的人员可以获得初始产生的“数字货币”作为回报。最早的ICO项目可追溯到2013年7月的Mastercoin项目,该项目通过比特币进行ICO众筹,生成对应的Mastercoin代币并分发给众筹参与者。知名的以太坊项目的初始研发资金也是通过ICO的方式筹得的。然而,因为不易监管,ICO融资方式极易被投机者利用。特别是2017年以来,国内ICO融资项目数量迅速爆发,2017年1至4月上线8个ICO项目,5月上线9个,6月上线了27个,导致ICO充斥着投机行为和泡沫,监管层给予了高度关注,9月4日,央行、银监会等七部门发布通告,宣布任何组织和个人不得非法从事代币发行融资活动。随后,主要“数字货币”均应声迅速下跌。ICO的狂热从侧面反映了区块链产业受关注的程度,但想借助区块链热度进行短期投机炒作甚至诈骗的行为注定不能长久,只有踏踏实实地进行区块链技术创新,才能迎来真正的技术爆发期。

下面来看看大数据平台所展示的一些与区块链技术相关的信息。据统计,在谷歌趋势中,区块链技术按区域显示的搜索热度排行中,印度排名第一,然后依次是澳大利亚、印度尼西亚、加拿大、英国和美国。这个排名与国家人口数量有关,但也与国家对区块链技术的关注度有很大的关系。谷歌趋势目前并没有中国的数据,因此暂时不清楚中国和其他国家对于区块链技术的搜索热度对比。但为了探究区块链在中国的热度以及趋势情况,通过与谷歌趋势类似的百度指数平台进行分析,发现在国内区块链的热度从2015年8月开始,一直呈上升趋势,这或许与2015年10月首届全球区块链峰会的召开和宣传有关,之后更多的人接触和关注区块链这一新技术。到2016年1月,中国央行召开研讨会,讨论采用区块链技术发行“数字货币”的可能性,推动区块链的百度指数继续显著提升。直至2016年6月,由于全球闻名的、也是当时最大的众筹项目DAO被黑客攻击而被迫采用通过硬分叉的措施解决这一事件带来的损失,导致区块链的价值和安全性受到了大众质疑,相对应的百度指数出现了明显下滑。而到2016年8月,工信部发布区块链发展白皮书,肯定了区块链技术的价值,指数又开始再次反弹,并稳步提升。2017年,随着全球区块链金融(杭州)峰会、工信部首届中国区块链开发大赛等大型区块链活动的举办,区块链的热度持续攀升。

通过以上一系列数据分析,可以发现,在短短的两三年时间内,区块链这一新兴技术发展得如此之快,态势如此之猛烈。这不禁让人联想到了前些年的互联网,互联网实现了信息传播和分享,而区块链技术宣告了互联网从传递信息的信息互联网向转移价值的价值互联网的进化。

1.4.2 区块链政府规划

随着区块链技术的不断发展,各国对区块链的认知程度逐渐提高,政府相关部门纷纷对区块链技术予以关注、探讨和推动,并推出相应发展规划,如图1.9所示。

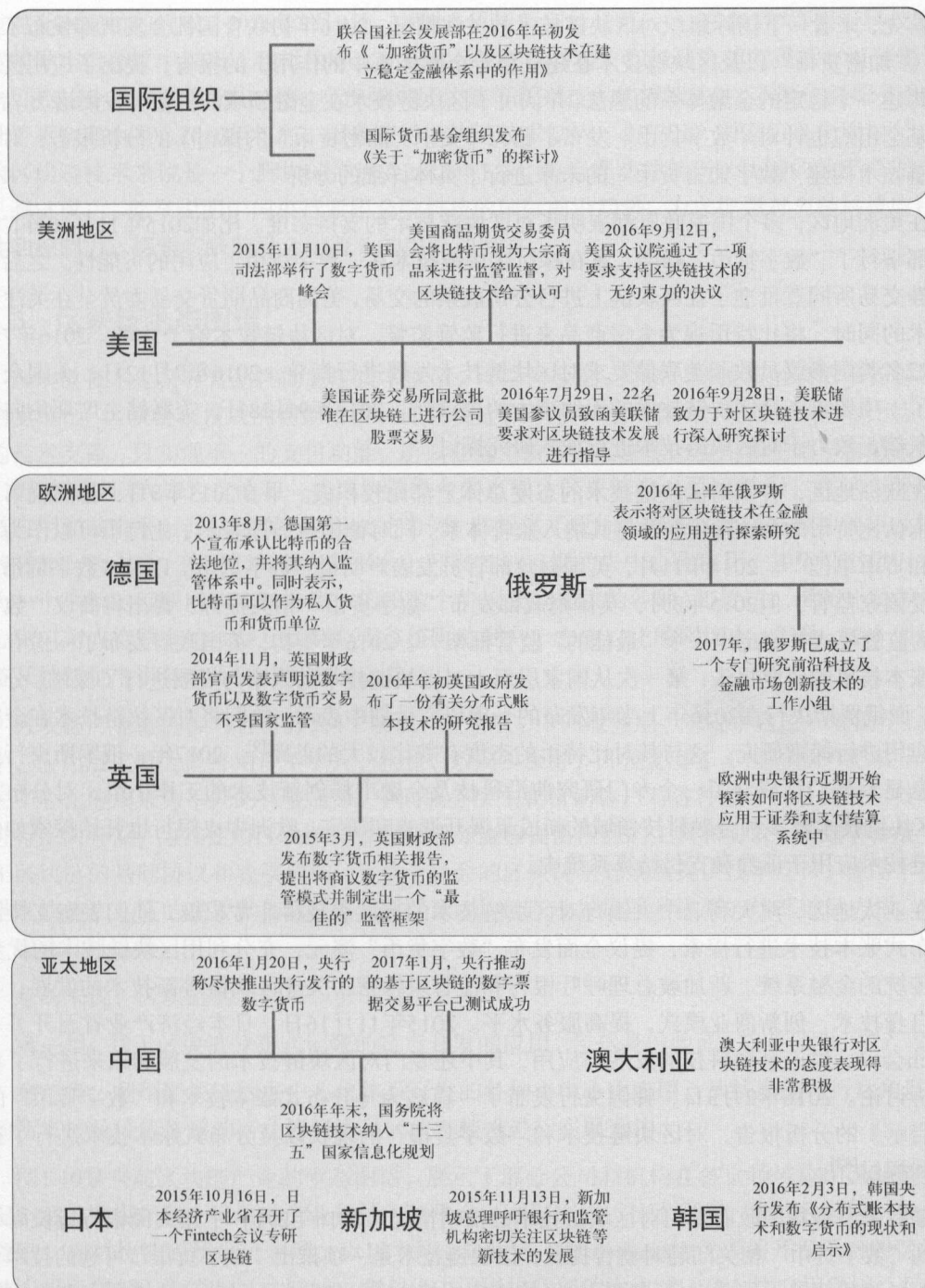


图1.9 各国区块链发展战略与规划

首先,来看一下国际组织对区块链技术的关注情况。2016年初联合国社会发展部发布了一篇题为《“加密货币”以及区块链技术在建立稳定金融体系中的作用》的报告,提出了应用区块链技术构建一个稳定的金融体系的想法,并认可了区块链技术在金融领域的价值和发展潜力。后来,国际货币基金组织也针对“数字货币”发布了题为《关于“加密货币”的探讨》的分析报告,对使用区块链技术构建“数字加密货币”的未来进行了具体详细的分析^[42]。

在美洲地区,多个国家政府都表明了对区块链技术支持的态度。比如2015年11月10日,美国司法部举行了“数字货币”峰会,意在探讨区块链技术在“数字货币”应用的可能性。之后,美国证券交易所同意批准了在区块链上进行公司股票的交易,美国商品期货交易委员会在关注区块链技术的同时,将比特币视为大宗商品来进行监管监督,对区块链技术给予认可。2016年7月29日,22名美国参议员致函美联储要求对区块链技术发展进行指导,2016年9月12日,美国众议院通过了一项要求支持区块链技术的无约束力的决议^[43]。2016年9月28日,美联储主席耶伦向外透露美联储正致力于对区块链技术进行深入研究探讨^[40]。

在欧洲地区,各国对区块链技术的态度总体上都比较积极。早在2013年8月,德国就第一个宣布承认比特币的合法地位,并将其纳入监管体系。同时德国政府还表示,比特币可以作为私人货币和货币单位^[41]。2014年11月,英国财政部官员发表声明说“数字货币”以及“数字货币”交易不受国家监管,但2015年3月,英国财政部发布“数字货币”相关报告,提出将商议“数字货币”的监管模式并制定出一个“最佳的”监管框架^[42]。2016年年初,英国政府发布了一份有关分布式账本技术的研究报告,第一次从国家层面上对区块链技术的未来与发展进行了探讨、分析和建议。而俄罗斯央行在2016年上半年发布的一项研究计划中表示,他们将对区块链技术在金融领域的应用进行探索研究,这与其对比特币的态度有着比较大的差距^[9]。2017年,俄罗斯央行发布的信息显示他们已成立了一个专门研究前沿科技及金融市场创新技术的工作小组,对分布式账本、区块链技术及多种金融科技领域的新成果展开调查和研究。欧洲中央银行也开始探索如何将区块链技术应用与证券和支付结算系统中。

在亚太地区,澳大利亚中央银行对区块链技术的态度表现得非常积极,他们表态支持银行对分布式账本技术进行探索,提议全面发布“数字货币”澳元,充分利用区块链技术的优势来革新传统的金融系统。新加坡总理呼吁银行和监管机构密切关注区块链等新技术的发展,不断改进自身技术,创新商业模式,提高服务水平。2015年11月16日,日本经济产业省召开了一个Fintech会议,讨论金融科技的发展与应用,其中还专门对区块链技术的发展和未来进行了具体的分析讨论。2016年2月3日,韩国央行发布了一篇题为《分布式账本技术和“数字货币”的现状和启示》的分析报告,对区块链技术和“数字货币”的现状以及分布式账本技术进行了积极研究和探讨^[42]。

最后来看看我国政府部门对区块链技术的推动情况。2016年2月,中国人民银行行长周小川在谈到“数字货币”相关问题时就曾提及,区块链技术是一项推出“数字货币”可选的技术,并提到人民银行部署了重要力量研究探讨区块链应用技术^[44]。他认为,目前区块链还存在着比较大的问题,比如区块链技术需要巨大的计算资源以及存储资源,同时区块产生的时间太长,还没办

法应对金融交易的规模。2016年9月9日,中国人民银行副行长范一飞在2015年度银行科技发展奖评审领导小组会议中提出,各机构应主动探索系统架构转型,积极研究建立灵活、可延展性强、安全可控的分布式系统架构,同时应加强对区块链等新兴技术的持续关注,不断创新服务和产品,提升普惠金融水平。2016年年底,国务院将区块链技术纳入“十三五”国家信息化规划^[45],对于国内区块链技术来说是一个巨大的推进。2017年1月,央行推动的基于区块链的数字票据交易平台已测试成功。此举说明中国央行紧跟金融科技的国际前沿趋势,力求把握对金融科技应用的前瞻性和控制力、探索实践前沿金融服务的决心和努力。

1.4.3 区块链生态图谱

区块链技术是具有普适性的底层技术框架,可以为金融、经济、科技甚至政治等各领域带来深刻变革^[2]。区块链在发展的初期阶段,即区块链1.0阶段,主要作为“数字货币”(比特币)体系的技术支撑,只实现单一的支付功能,所以在这个阶段,区块链的应用和基础平台是紧密耦合的。但随着以以太坊为首的新一代区块链平台的出现,区块链进入了2.0阶段,在这个阶段,区块链应用和基础平台开始解耦。以以太坊为例,其提供了更加完善的区块链基础协议以及图灵完备的智能合约语言,使我们可以在其平台上开发各种各样的去中心化应用。甚至可以将以以太坊类比为一个新的互联网TCP/IP协议,依赖这个协议及其提供的各种API接口,帮助开发者开发去中心化应用或将原有的一些互联网应用移植再造到一个去中心化的网络中。于是,整个区块链产业链开始衍生出了各个不同的生态层次^[46]。

区块链产业链的参与者可分为4个层次:应用层、中间服务层、基础平台层和辅助平台层^[43]。其中,应用层主要为最终用户(个人、企业、政府)服务,开发者基于不同的用户需求开发不同的去中心化应用来为不同的行业服务;中间服务层主要帮助客户对各种基于区块链底层技术的应用进行二次开发,为其使用区块链技术改造业务流程提供便捷的工具和协议;基础平台层主要聚焦于区块链的基础协议和底层架构,为整个社会的区块链生态发展提供技术支持;辅助平台层并不是区块链产业链的主要参与者,但其同样是区块链产业发展非常重要的外部辅助力量,包括基金、媒体和社区等。

总的来说,区块链的应用可以分为两类。

第一类,基于区块链分布式记账的特点开发的应用,包括身份验证、权益证明、资产鉴证等。

第二类,利用区块链的去中心化体系开发的去中心化应用,从技术的可行性角度来看,目前所有涉及价值传递的行业皆可通过区块链技术进行底层重构。

图1.10是当前区块链产业的生态图谱,展示了部分公司和机构在各自领域应用区块链技术的情况。总的来说,整个区块链产业包括底层平台、上层应用、技术研究、媒体及社区、投资等生态领域。在区块链底层平台领域,以以太坊、Fabric、Hyperchain为代表的开发平台对区块链底层技术进行革新,为基于区块链的去中心化应用提供底层技术支撑。在上层应用领域,开发者在各行各业展开了应用场景探索,如以Ripple、Circle公司为代表的金融服务领域,以Factom公司为

从而实现了效率与灵活的完美平衡。由于参与方存在互信问题,传统的票据流通审核繁琐,变现困难,难以实现互通互利。通过将票据信息登记在区块链平台上,利用区块链扩展成本低、交易步骤简化的特性,将票据转变为客户可持有、可流通、可拆分、可变现的具有一定标准化程度的数字资产。

1.5.2 供应链金融

传统的供应链金融平台一般由单个金融机构主导,难以实现同业间的扩展和推广。区块链技术让参与方只需专注于业务系统对接区块链平台即可,可实现全行业的快速覆盖。供应链上企业之间的贸易信息、授信融资信息,以及贸易过程中涉及的仓储、物流信息均登记在区块链上,且信息不可篡改,保证了资产的真实有效,降低了企业融资成本和银行授信成本。跨机构信息通过区块链的共识机制和分布式账本保持同步,通过访问任意一个节点即可获取完整的交易数据,打破信息孤岛。机构通过访问内部区块链节点即可获得完整的交易数据,增强企业间的信用协作。通过将应收账款、承兑汇票、仓单等资产凭证记录在区块链上,并支持转让、质押等相关操作,实现了资产数字化,并通过区块链构造了一个数字化的、可以点对点传输价值的信用系统,实现了区块链上的价值传输。这一可信赖的价值传输系统既提高了需求方的融资能力,又提高了供应方的监管能力,为金融系统健康稳定提供了根本保障。通过智能合约控制供应链流程,减少人为交互,提升产业效率。无需中心平台审核确认,通过传感器探测真实仓储、物流信息,使用无线通信网络发送可信数据到区块链验证节点,保证满足合约条件时,自动触发相关操作,减少操作失误。

1.5.3 应收账款

传统的应收账款通过线下交易确认的方式完成,而伪造交易、篡改应收账款信息等风险的存在降低了交易参与方的信任感。将应收账款的全流程操作通过区块链平台进行,实现了应收账款交易的全程签名认证并且不可抵赖,同时使用智能合约实现权限和状态控制,使得应收账款更加安全可控,构建了高度可信的交易平台。应收账款交易流程中参与方众多,业务复杂,面对传统应收账款的融资申请,金融机构需要进行大量的贸易背景审查。区块链平台通过时间戳来记录整个应收账款的生命周期,从而使得所有的市场参与者都可以看到资金流和信息流,排除了票据造假的可能性。传统的应收账款由于存在互信问题,在交易市场上流通困难。应收账款以数字资产的方式进行存储、交易,不易丢失和无法篡改的特点使得新的业务模式可以快速推广,在提高客户资金管理效率的同时降低使用成本,并在不同企业间形成互信机制,使得多个金融生态圈可以通过区块链平台互通互利,具有良好的业务价值和广阔的发展空间。

1.5.4 数据交易

数据作为特殊商品具有独特性,存在被复制、转存的风险,按照商品流通中介模式建立的数

据中介平台构成了对数据交易双方权益的潜在威胁,变成了数据交易的一个障碍。只有建立符合数据特性的信息平台,通过技术机制而不是仅凭承诺来保障数据的安全和权益,做到让数据交易双方真正放心,才能加速数据的顺畅流动。通过区块链技术对数据进行确权,能够有效保障数据所有方的权益,杜绝数据被多次复制转卖的风险,把数据变成受保护的虚拟资产,对每笔交易和数据进行确权 and 记录。利用区块链的可追溯和不可篡改等特性,可以确保数据交易的合规、有效,激发数据交易的积极性,促成数据市场的规模性增长。

1.5.5 债券交易

债券业务是需要多家机构共同参与的一项业务,在其发行、交易等流程中,各机构之间需要通过传统的邮寄或者报文转发的形式进行信息的同步与确认。债券发行交易如果通过中心化系统实现,可能会存在人工操作性失误或恶意篡改的风险。使用区块链技术之后,系统可以由区块链底层来保证数据的同步与一致,降低不同机构系统之间对接的时间、人力和资金成本,从依靠基于业务流的低效协同升级为不依靠任何中介而由平台保证基本业务流程的低成本、高效率、高可信协作系统。而且传统的中心化系统很多信息都封闭在机构内部,无法对外部系统进行及时、有效的监管,监管会存在盲区。利用区块链技术,监管机构以节点的形式加入区块链,实时监控区块链上的交易。同时,智能合约使得债券在整个生命周期中具备限制性和可控制性,也可以有效提高监管效能。由于区块链的数据完整和不可篡改性,对任何价值交换历史记录都可以追踪和查询,能够清晰查看和控制债券的流转过程,从而保证债券交易的安全性、有效性和真实性,有效防范市场风险。同时,基于区块链技术可避免第三方机构对账清算的工作,从而有效提升债券交易的清算效率。

1.5.6 大宗交易

基于区块链技术的大宗交易平台,可以实现各清算行之间大宗交易的实时清算,提高大宗交易效率,为业务开展提供便利。智能合约控制大宗交易流程,减少人为交互,提升处理效率。无需中心平台审核确认,保证报价满足撮合条件时,自动触发相关操作,减少操作失误。交易所和清算所可以互为主备,负责所有交易数据定序广播,发起共识。实时灾备容错,发生重大故障可以秒级切换主节点。接入节点发生故障,通过内置算法快速恢复历史数据,避免交易数据丢失。会员和银行接入端独立处理查询,数据实时同步,减轻主节点压力。监管节点实时获取相关交易数据,监管机构对大宗交易进行实时监控。

1.5.7 其他场景

区块链是一种可以进行价值传输的协议,除了上述场景之外,还可应用于其他一切与价值转移有关的场景,如数字版权、公证、身份认证、社会公益,等等。

在消费金融领域,阳光保险公司用区块链技术作为底层技术架构推出了“阳光贝”积分,用

户在享受普通积分功能的基础上,还能以“发红包”的形式将积分向朋友转赠,并可与其他公司发行的积分进行互换^[51]。

在数字版权领域,知识产权侵权现象严重,基于区块链技术可以通过时间戳、哈希算法对作品进行确权,证明知识产权的存在性、真实性和唯一性,并可对作品的全生命周期进行追溯,极大地降低了维权成本^[52]。

在医疗领域,患者私密信息泄露情况时有发生,2015年4月,Factom宣称与医疗记录和服务方案供应商Healthnautica展开合作,研究运用区块链技术保护医疗记录以及追踪账目,为医疗记录公司提供防篡改数据管理^[53]。

在教育领域,目前学生信用体系不完整,无历史数据信息链,这导致政府和用人企业无法获得完整、有效的信息,利用区块链技术对学生的学历信息进行存储,可以解决信息不透明及容易被篡改的问题,有利于构建良性的学生信用体系^[54]。

在社会公益领域,慈善机构想要获得群众的支持,就必须具有公信力,而信息的透明则是必要条件之一,蚂蚁金服等公司已开始把区块链技术应用用于公益捐赠平台,这为加速公益透明化提供了一种可能^[55]。区块链技术也可用于政府信息公开领域,帮助政府部门实施公共治理及服务创新,提升政府部门的效率及效力^[56]。

关于区块链的应用场景还有很多,区块链的未来存在着无限的可能,这需要更多优秀的公司、企业和人才加入到区块链技术的探索队伍中,这样才能使区块链技术得到更快、更好的发展。人们有理由期待在区块链技术的范式下,又一次“大航海时代”的来临,将给各行各业和社会带来一次重构。

1.6 区块链主流平台

本节将对当前主流的区块链平台进行简介和对比分析。

比特币(Bitcoin)是第一个区块链应用,使用工作量证明机制来达成网络节点的共识,由于比特币网络任何人都可以加入,没有访问权限,因此它是一个公有链,不支持智能合约,但是可以支持一些图灵不完备的编程脚本来进行一些简单的操作编程。其公网TPS小于7。

以太坊(Ethereum)是一个图灵完备的区块链一站式开发平台,采用多种编程语言实现协议,采用Go语言写的客户端作为默认客户端(即与以太坊网络交互的方法支持其他多种语言的客户端)。基于以太坊平台之上的应用是智能合约,这是以太坊的核心。智能合约配合友好的界面,外加一些额外的小支持,可以让用户基于合约搭建各种各样的DApp应用,这样使得开发人员开发区块链应用的门槛大大降低。目前以太坊正在正式运行1.0版本,使用PoW共识机制,公网TPS约为100。

Hyperledger Fabric是Linux基金会成立的Hyperledger联盟所推出的一个孵化中的项目,目前

正在构建标准化的数字账本^[57]，它旨在使用区块链技术帮助新创公司、政府和企业联盟之间减少工作的花费和提高效率。因此，它不是面向公众的，而是服务于公司、企业、组织等联盟团体，属于联盟链。平台设计所使用的是Go语言，共识算法则是PBFT算法。同样，它也是支持智能合约编程的，在Fabric中有自己的学名，叫Chaincode。另外，Chaincode仅在验证节点上执行，且运行在被隔离的沙盒中，目前采用Docker作为执行Chaincode的容器^[58]。Fabric的公网TPS约为3000。

比特股（BitShares）是一个点对点的多态数字资产交易系统，是DPoS共识机制的鼻祖。它提供的BitUSD等锚定资产是虚拟货币历史上最重要的变革之一，能够极大地消除“虚拟货币”被人诟病的波动性大的问题。此外，比特股内置了强大的账户权限设定、灵活的多重签名方式、白名单等特性，足以满足企业级的功能定制需求^[59]。比特股区块链是一个公有链，其核心技术框架采用C++语言进行开发，公网TPS大于500。

公证通（Factom）利用比特币的区块链技术来革新商业社会和政府部门的数据管理和数据记录方式^[60]。利用区块链技术帮助各种应用程序的开发，包括审计系统、医疗信息记录、供应链管理、投票系统、财产契据、法律应用、金融系统等。建立在公证通基础之上的应用程序寻求能够直接利用区块链实现追踪资产和实现合约，而不用将交易记录写入区块链，公证通在自己的架构中记录条目。公证通中的政策和奖励机制与PoS机制有相似之处。与其不同之处在于，公证通中只有一小部分的用户权益能够被认可。只有已经提交到系统的权益有投票权，而可转移的Factoid权益没有投票权，避免了PoS机制的“股份磨损”和“没有人进行PoS”的问题^[59]。公证通的核心技术框架采用Go语言开发，TPS为27左右。

瑞波（Ripple）是世界上第一个开放的支付网络，它引入了一个共识机制RPCA，通过特殊节点的投票，在很短的时间内就能够对交易进行验证和确认^[61]。瑞波客户端不需要下载区块链，它在普通节点上舍弃掉已经验证过的总账本链，只保留最近的已验证总账本和一个指向历史总账本的链接，因而同步和下载总账本的工作量很小^[59]。瑞波核心技术框架采用C++语言进行开发，公网TPS小于1000。

未来币（Nextcoin, NXT）是第二代去中心化“虚拟货币”，它使用全新的代码编写，不是比特币的山寨币。它第一个采用100%的股权证明PoS算法，有资产交易、任意消息、去中心化域名、账户租赁等多种功能，部分实现了透明锻造功能。透明锻造机制使得每一个用户客户端可以自动决定哪个服务器节点能够产生下一个区块，让客户端可以直接将交易发送到这个节点，从而使得交易的时间达到最短。实时和高优先级的交易可以通过支付额外的费用来被优先处理^[59]。未来币的核心技术框架所采用的开发语言是Java，TPS小于1000。

Hyperchain是杭州趣链科技开发的一个满足行业需求的联盟区块链技术基础平台，通过整合并改进区块链开源社区和研究领域的最前沿技术，集成了高性能的可靠共识算法RBFT，兼容开源社区的智能合约开发语言和执行环境，同时在记账授权机制和交易数据加密等关键特性上进行了强化，并且提供了功能强大的可视化Web管理控制台对区块链节点、账簿、交易和智能合约等进行高效管理^[62]。Hyperchain与Fabric一样采用了模块化设计理念，分为共识算法、权限管理、

多级加密、智能合约引擎、节点管理、区块池、账本存储、数据存储8个核心模块，旨在服务于票据、存单、股权、债券、登记、供应链管理等数字化资产、金融资产商业应用，并且其系统吞吐量可达到每秒处理上万笔交易，这在当前的区块链平台中是首屈一指的。

表1.3中列出了各个平台所使用的共识机制、所属区块链类型、平台开发所采用的语言、是否支持智能合约以及每秒事务处理量（TPS）性能指标，以供读者进行更直观的统计和对比。

表1.3 区块链平台比较

平 台	共识机制	类 型	语 言	智能合约	TPS
Bitcoin	PoW	Public	C++	N/A	<7
Ethereum	PoW&PoS	Public	Go	Yes	约100
Hyperledger Fabric	PBFT	Consortium	Go	Yes	约3000
BitShares	DPoS	Public	C++	N/A	>500
Factom	PoS	Public	Go	N/A	约27
Ripple	RPCA	Public	C++	N/A	<1000
NXT	PoS	Public	Java	N/A	<1000
Hyperchain	RBFT	Consortium	Go	Yes	>10 000

从上面的平台介绍和对比中可以看出，当前区块链平台使用的共识算法各有不同，对于不同的应用场景，相应的共识机制有各自的优点和不足。平台类型主要是公有链和联盟链这两种，私有链应用较少。平台设计所使用的编程语言主要是Go和C++，因为区块链网络所处环境是一个分布式网络，需要高并发和高效率的操作执行。是否支持智能合约与每个平台所面向的场景和所提供的服务有关，比如以太坊、Hyperledger Fabric、Hyperchain等作为底层平台，一般都需要提供智能合约功能，而对于某些应用平台，智能合约则不一定是必需的。区块链平台的性能则随着区块链技术的发展在不断地提升，在某些应用场合已基本满足商业应用的要求，其中Hyperchain平台的TPS已达到了10000，在区块链性能方面具有显著优势。

1.7 本章小结

本章对区块链技术进行了全景分析，介绍了区块链的基础知识和发展历程，对其关键技术和特性进行了详细的讲解，并结合时代背景分析了区块链的产业现状，选取了一些典型的应用场景进行阐述，最后对当前的区块链主流平台进行了介绍与对比，使读者对区块链技术有一个初步的了解和认识，为之后的进阶和实战打下基础。

参考文献

- [1] 林晓轩. 区块链技术在金融业的应用[J]. 中国金融, 2016(8): 17-18.
- [2] 袁勇, 王飞跃. 区块链技术发展现状与展望[J]. 自动化学报, 2016(4): 481-494.

- [3] Grinberg R. Bitcoin: An Innovative Alternative Digital Currency[J]. Social Science Electronic Publishing, 2011.
- [4] 端宏斌. 比特币悖论[J]. 中国经济和信息化, 2013(11): 27-28.
- [5] Nakamoto, Satoshi. Bitcoin: A peer-to-peer electronic cash system. 2008: 28.
- [6] Irni E K, Corina S, Sarah C, et al. Exploring Motivations among Bitcoin Users[J]. CHI Confernece Extended Abstracts on Human Factors in Computing Systems. 2016(5): 2872.
- [7] Antonopoulos, Andreas M. Mastering Bitcoin: unlocking digital cryptocurrencies (精通比特币). O'Reilly Media, Inc, 2014.
- [8] 区块链+: 无中介驱动的世界[EB/OL]. <http://36kr.com/p/5053134>. Html, 2016.
- [9] 工信部-中国区块链技术和应用发展白皮书(2016) [EB/OL]. <https://img2.btc123.com/file/0/chinabolck-chaindevwhitepage2016.pdf>, 2016.
- [10] Raval, Siraj. Decentralized Applications: Harnessing Bitcoin's Blockchain Technology. O'Reilly Media, Inc. pp. 1, 2. ISBN 1491924527. Retrieved 6 November 2016.
- [11] Swan, Melanie. Blockchain: Blueprint for a new economy. (区块链: 新经济蓝图) O'Reilly Media, Inc., 2015.
- [12] Pilkington M. Blockchain technology: principles and applications[J]. Research Handbook on Digital Transformations, edited by F. Xavier Olleros and Majlinda Zhegu. Edward Elgar, 2016.
- [13] Wikipedia, 区块链, [EB/OL]. <https://zh.wikipedia.org/zh-hans/区块链>, 2017.
- [14] Croman K, Decker C, Eyal I, et al. On scaling decentralized blockchains[C]//Proc. 3rd Workshop on Bitcoin and Blockchain Research. 2016.
- [15] 区块链技术演进史 [EB/OL]. <http://tech.hexun.com/2016-04-25/183507891.html>, 2016.
- [16] Lamport L, Shostak R E, Pease M C, et al. The Byzantine Generals Problem[J]. ACM Transactions on Programming Languages and Systems, 1982, 4(3): 382-401.
- [17] Koblitz N. Elliptic curve cryptosystems[J]. Mathematics of Computation, 1987, 48(177): 203-209.
- [18] Miller V S. Use of Elliptic Curves in Cryptography[C]. International cryptology conference, 1985: 417-426.
- [19] Chaum D, Fiat A, Naor M, et al. Untraceable electronic cash[C]. International cryptology conference, 1990: 319-327.
- [20] Lamport L. A theorem on atomicity in distributed algorithms[J]. 1990, 4(2):59-68.
- [21] Haber S, Stornetta W S. How to time-stamp a digital document[J]. Journal of Cryptology, 1991, 3(2): 99-111.
- [22] Johnson D H, Menezes A, Vanstone S A, et al. The Elliptic Curve Digital Signature Algorithm (ECDSA)[J]. International Journal of Information Security, 2001, 1(1): 36-63.
- [23] Back A. Hashcash-A Denial of Service Counter-Measure[C]// USENIX Technical Conference. 2002.
- [24] Dai, Wei. B-money[EB/OL]. <http://www.weidai.com/bmoney.txt>, 1998.
- [25] Finney H. RPOW-Reusable Proofs of Work [EB/OL]. <http://cryptome.org/rpow.html>, 2004.
- [26] 李鸿. 一种基于椭圆曲线的部分盲签名方案[J]. 宿州师专学报, 2004(1):89-91.
- [27] Adishesu H, T. V Lakshman. The Internet Blockchain: A Distributed, Tamper-Resistant Transaction Framework for the Internet. ACM. 2016(11). 205.
- [28] Buterin, Vitalik. Ethereum: A next-generation smart contract and decentralized application platform[EB/OL]. <https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-White-Paper> (2014).
- [29] 区块链核心技术演进之路-共识机制演进(1) [EB/OL]. <http://www.8btc.com/blockchain-tech-consensus-mechanism>, 2016.
- [30] 杨涛. 浅析比特币的非货币属性[J]. 时代金融, 2014(1):93-94, 96.

- [31] King S, Nadal S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake[J]. Self-published paper, 2012(8), 19.
- [32] Daniel L, Charles H, Stan L. Bitshare:A Peer-to-Peer Polymorphic Digital Asset Exchange[J]. Self-published paper, 2013(9), 11.
- [33] Castro M, Liskov B. Practical Byzantine fault tolerance[C]//OSDI. 1999, 99: 173-186.
- [34] Bitcoinmining Article [EB/OL]. <https://www.bitcoinmining.com/bitcoin-mining-pools>, 2015.
- [35] Szabo N. The idea of smart contracts, 1997[J]. 1997.
- [36] Delmolino K, Arnett M, Kosba A E, et al. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab[J]. IACR Cryptology ePrint Archive, 2015: 460.
- [37] Kosba A, Miller A, Shi E, et al. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts[J]. University of Maryland and Cornell University, 2015.
- [38] Luu L, Chu D H, Olickel H, et al. Making smart contracts smarter[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016: 254-269.
- [39] Juels A, Kosba A, Shi E. The ring of gyges: Using smart contracts for crime[J]. aries, 2015, 40: 54.
- [40] 从无大同世界, 区块链去中央化不是无国界, [EB/OL]. <http://server.zol.com.cn/630/6304620.html?keyfrom=front>, 03, 2017.
- [41] 区块链的产业生态现状, [EB/OL]. http://www.xianjichina.com/news/details_28596.html, 03, 2017.
- [42] 唐文建, 吕雯, 黄浩. 区块链如何重新定义世界, 2016.
- [43] 2016—2020年区块链技术深度调研及投资前景预测报告, 2016.
- [44] 中国区块链应用研究中心. 图解区块链[M]. 首都经济贸易大学出版社. 2016:50-51.
- [45] 国务院. 国务院关于印发“十三五”国家信息化规划的通知[R]. 2016.
- [46] 区块链研究报告: 应用场景、商业模式及产业链[EB/OL]. <http://3g.ifeng.com/news/sharenews.f?aid=110489342&mid=,06>, 2016.
- [47] 麦肯锡报告, 区块链: 银行业游戏规则颠覆者[EB/OL]. <http://www.mckinsey.com.cn/wp-content/uploads/2016/05/%E5%8C%BA%E5%9D%97%E9%93%BE.pdf>, 2016.
- [48] 高盛. 区块链技术每年能够为资本市场节约60亿美元[R]. 2016.
- [49] 张健. 区块链: 定义未来金融与经济新格局, 2016.
- [50] 供应链区块链融资Skuchain 无纸化时代的到来 [EB/OL]. <http://chainb.com/?P=Cont&id=1336>, 07, 2016.
- [51] 阳光保险成为国内首家推出区块链应用的金融企业 [EB/OL]. <http://about.sinosig.com/common/news/html/144363.html>, 2016.
- [52] 知识产权服务遇到区块链: 天作之合[EB/OL]. <http://www.z3z4.com/a/4lyft1s>, 09, 2016.
- [53] 浅谈区块链+医疗[EB/OL]. <http://www.onesheng.cn/news/118016.html>, 11, 2016.
- [54] 区块链技术及其在教育领域的初步探索[EB/OL]. <http://www.ndedu.gov.cn/html/xwzx/jyxxh/2017/01/10/64a0c253-ffc0-4dbe-9654-1077c36d5e40.html>, 2016.
- [55] 区块链能否让“郭美美”远离慈善[EB/OL]. <http://www.it-times.com.cn/sbdj/57659.j.html>, 2016.
- [56] 区块链技术对于政府治理创新的影响[EB/OL]. <http://www.echinagov.com/knowledge/CIO/46723.html>, 2016.
- [57] Sarah Underwood. Blockchain Beyond Bitcoin[J]. Communications of the ACM, 2016(11):16.
- [58] Cachin C. Architecture of the Hyperledger Blockchain Fabric[J]. 2016.
- [59] 区块链主流开源技术体系介绍[EB/OL]. <http://blog.csdn.net/elwingao/article/details/52679475>, 2016.

- [60] Snow P, Deery B, Lu J, et al. Factom: Business Processes Secured by Immutable Audit Trails on the Blockchain[J]. 2014.
- [61] Moreno-Sanchez P, Zafar M B, Kate A. Listening to whispers of ripple: Linking wallets and deanonymizing transactions in the ripple network[J]. Proceedings on Privacy Enhancing Technologies, 2016(4): 436-453.
- [62] 趣链科技. Hyperchain白皮书. 2016.

第二部分

开源区块链平台

- 第2章 以太坊深入解读
- 第3章 以太坊应用开发基础
- 第4章 Hyperledger Fabric 深入解读
- 第5章 Hyperledger Fabric 应用开发基础

以太坊是一个图灵完备构建去中心化应用的开源平台，目前已经积累了大量的开发者。本章将由浅入深地对以太坊进行讲解，主要包括：以太坊的相关概念，如节点、挖矿、账户、gas、消息等；以太坊核心原理，包括共识机制、EVM以太坊虚拟机、数据存储和加密算法；以太坊智能合约，包括智能合约的语法结构、编译部署与测试；重大事件剖析，包含以太坊ICO、The DAO事件和目前以太坊存在的主要问题，如共识效率问题、隐私保护问题等。

以太坊是一个全新的区块链应用平台，被认为是区块链2.0。以太坊允许任何人通过智能合约在平台上建立和使用基于区块链技术的去中心化应用DApp。以太坊的核心理念是内置图灵完备编程语言的区块链，“图灵完备”的意思在于一切可计算的问题都能通过计算解决。建立这种图灵完备的基础就是以太坊虚拟机（Ethereum Virtual Machine, EVM）。EVM类似于Java虚拟机（JVM），编译后基于字节码运行，开发时则可以使用高级语言实现，编译器会自动转化为字节码。

基于以太坊的应用在进行交易和运行时，需要提供一种事先定义的业务逻辑来满足规范化的运行，这种机制在区块链中称为智能合约。一个合约就像一个自动化的代理，当条件满足时，智能合约就会自动运行一段特定的代码，完成指定的逻辑。关于智能合约将在2.3节中做详细的讲解。

2.1 以太坊基础入门

本节将从以太坊的发展历史、基本概念、客户端种类、账户管理方法和以太坊网络这几个方面对以太坊进行介绍，以帮助读者直观全面地了解以太坊，为基于以太坊的项目开发打下基础。

2.1.1 以太坊发展历史

2013年年末，以太坊创始人Vitalik Buterin发布了以太坊白皮书，在全球的密码学货币社区召集了一批赞同以太坊理念的开发者，启动了以太坊项目。

2014年3月初，以太坊发布了第三版测试网络（POC3）。4月，Gavin Wood发布了以太坊黄皮

书，将以太坊虚拟机等重要技术规范化。6月，团队发布POC4，并快速向POC5前进。在这期间，团队还决定将以太坊发展为一个非营利组织。

2015年7月，以太坊发布了Frontier版本，其最主要的功能就是进行以太坊挖矿。有了挖矿的功能，开发者就可以在以太坊区块链中测试去中心化应用。Frontier版本的以太坊客户端有多种语言实现，但是只有命令行界面，没有提供图形化界面，所以该阶段的使用者主要是开发者和研究人员。

2016年3月，以太坊发布了Homestead版本。在本书编写阶段以太坊正处于Homestead版本。这表明以太坊网络已经平稳运行。在此阶段，以太坊提供了图形界面的以太坊钱包，除开发人员以外的非技术人员也可以进行简单的操作。

Metropolis版本发布日期尚未确定。在Metropolis版本中，以太坊社区将最终正式发布一个为非技术用户设计的Mist浏览器，Mist可以认为是以太坊钱包的升级版。随着第三方开发者为普通用户开发的去中心化应用逐渐增多，以太坊将不仅是一个开发平台，也是一个应用市场。

Serenity第四阶段版本发布日期尚未确定。在Serenity版本中，以太坊将共识机制从PoW工作量证明转换到PoS权益证明。PoW是对计算能力和电力资源的严重浪费，而PoS将提高区块产生效率。转换到PoS算法以后，前三个阶段所需要的挖矿将被终止，新发行的比特币也将大为降低，甚至不再增发新币。

2.1.2 以太坊基本概念

以太坊由大量的节点组成，节点有账户与之对应，两个账户之间通过发送消息进行一笔“交易”。交易里携带的信息和实现特定功能的代码称为智能合约，运行智能合约的环境是以太坊虚拟机。以太坊虚拟机运行在每个节点中，交易需要有节点参与，通过重复哈希运算来产生工作量，这些节点称为矿工，计算的过程称为挖矿。交易的计算是要付出费用的，这些费用就称为gas。在以太坊中，gas是由比特币转换生成的。比特币是以太坊上用来支付交易手续费和运算服务的媒介，消耗的gas用于奖励矿工。基于以上智能合约代码和以太坊平台的应用叫作去中心化应用。以太坊的基本概念包括以下几个方面。

(1) 节点。通过节点可以进行区块链数据的读写。目前以太坊上的很多应用都是基于公有链的，所以每一个节点都拥有相同的地位和权利，没有中央服务器，每个节点都可以加入网络，读写以太坊中的数据。节点之间使用共识机制来确保数据交互的可靠性和正确性。单独的一个节点也可以搭建私有链，几个相互信任的节点可以搭建联盟链。

(2) 矿工。矿工是指通过不断重复哈希运算来产生工作量的网络节点。矿工的任务是计算数学难题，并将计算结果放入新的区块中。矿工之间是竞争关系，最先计算出结果的节点，将向全网进行广播，当结果被确认后，新生成区块所包含的奖励将会给该节点，存入比特币地址中。该节点所包含的比特币可以作为下次发起交易的资产。

(3) 挖矿。在以太坊中，发行以太币的唯一途径是挖矿。挖矿过程也保证了区块链中交易的验证与可靠性。挖矿是一个需要消耗大量算力和时间的工作，并被限制在一定的时间期限内，同时挖矿难度可以动态调整。简单来说，挖矿的过程就是矿工寻找一个随机数进行SHA256计算哈希值，如果计算后的哈希值满足一定的条件，比如前60位为0或小于等于某个预先的随机数(Nonce)，那么这个矿工就赢得了创建区块的权利。

(4) 账户。以太坊中包含两类账户：外部账户和合约账户。外部账户由公私钥对控制。合约账户则在区块链上唯一标识了某个智能合约。两类账户都包含了以太币余额，能发送交易。每个账户的地址长度为20字节，有一块持久化内存区域被称为存储区(storage)，其形式为键值对，键和值的长度均为32字节。重要的是，外部账户的地址是由公钥决定的，合约账户的地址是在部署合约的时候确定的，当合约账户接收到一笔合法的交易后，就会执行里面包含的合约代码。所以两类账户最大的区别是：合约账户存储了代码，外部账户则没有。

(5) gas。以太坊上的每一笔交易都有矿工的参与，且都需要支付一定的费用，这个费用在以太坊中称为gas。gas的目的是限制执行交易所需的工作量，同时为执行交易支付费用。合约的代码在EVM上运行时，gas会按照既定的规则逐渐消耗。gas价格是由交易创建者设置的，交易费用=gas price * gas amount。如果执行结束后还有gas剩余，这些gas将会返还给发送者账户，而消费的gas则被当作奖励，发放到矿工账户。

(6) EVM。以太坊虚拟机是以太坊中智能合约的运行环境，并且是一个沙盒，与外界隔离。智能合约代码在EVM内部运行时，是不能进行网络操作、文件I/O或执行其他进程的。智能合约之间也只能进行有限的调用，这样保证了合约运行的独立性，并尽可能提高了运行时的安全性。

(7) 智能合约。合约是代码和数据的集合，存在于以太坊区块链的指定地址。合约方法支持回滚操作，如果在执行某个方法时发生异常，如gas消耗完，则该方法已经执行的操作都会被回滚。但是如果错误的交易一旦执行完毕，是没有办法篡改的。对于智能合约的详细开发细节请参考2.3节。

(8) 交易。在以太坊中，交易都是通过状态转移来标记的，状态由被称为“账户”的对象和两个账户之间的转移价值和信息状态转换构成。以太坊账户分为由公私钥控制的外部账户和由合约代码控制的合约账户。外部账户没有代码，用户通过创建和签名一笔交易从一个外部账户发送消息，合约账户收到消息后，合约内部代码会被激活，对内部存储进行读取和写入，或者发送消息，或者调用方法。

确定了账户后，即开始以太坊的交易。在以太坊中，“交易”是指存储从外部账户发出的消息的签名数据包，在交易过程中比较重要的是消息机制。以太坊的消息机制能够确保合约账户和外部账户拥有同等的权利，包括发送消息和创建其他合约。这使得合约可以同时由多个不同角色参与，共同签名来提供服务，而不需要关心合约的每一方到底是什么类型的账户。

每一笔交易的执行过程如下。

- (1) 检查交易是否有效，格式是否正确。
- (2) 计算交易所需费用，并判断交易发送者余额是否足够，如果足够，则从发送者账户中扣除交易费用。
- (3) 设定初始gas值。
- (4) 从发送者账户转移价值到接收者账户，若接收账户不存在则创建新账户并作为接收者。如果接收者是合约，则执行合约代码，直至代码运行完毕或者gas消耗完毕。

2.1.3 以太坊客户端

以太坊在发布之后，有多个语言版本的客户端，并且支持多平台、多语言，如go-ethereum、cpp-ethereum等。在以太坊进入Homestead阶段后，Go语言客户端占据了主导地位。同时官方开发的以太坊钱包逐步流行，本节将对主流的以太坊客户端进行介绍。

1. go-ethereum

go-ethereum客户端简称为geth，是一个完全的命令行界面，同时也是一个以太坊节点。通过安装和运行geth，可以实现搭建私有链、挖矿、账户管理、部署智能合约、调用以太坊接口等常用功能。下面是geth的常用命令行。命令行组成为geth --<命令>参数。

```
geth +
--datadir 指定数据存储位置（也是默认的私钥仓库位置）
--nodiscover 标示此节点私有，不被别人添加
--maxpeers 0 设置网络中可以被接入的最大节点数目，0代表不被其他节点接入
--rpc 允许节点的RPC接口打开（默认打开）
--rpcapi "db,eth,net,web3" 配置RPC接口允许访问的API（默认情况下，RPC接口是允许访问web3的API的）
--rpcport "8080" 设置geth端口号（默认为8080）
--rpccorsdomain "http://chriseth.github.io/browser-solidity/" 设置可以连接此节点的网址（并不是通配符）
--port "30303" 设置监听端口号，用于与其他节点进行连接
--identity "TestnetMainNode" 设置节点标识
--datadir<账户数据存放地址>
account new 用于创建一个新账户
account list 查询账户列表
init <genesis.json 文件路径> 根据genesis.json文件初始化创世区块
```

进入geth控制台后，有如下常用命令。

- > eth.accounts 查询账户列表
- > personal.listAccounts 查询账户列表
- > personal.newAccount() 创建新账户
- > personal.deleteAccount(addr,passwd) 删除账户
- > personal.unlockAccount(addr,passwd,time) 解锁账户，可进行交易操作
- > eth.sendTransaction({}) 发送交易

开启geth客户端如下：

```
➔ ~ geth --rpc --rpcapi="db,eth,net,web3,personal,web3" -
rpcaddr="0.0.0.0" --rpccorsdomain="*" --unlock '0' --password
~/Library/Ethereum/password --nodiscover --maxpeers '5' -
networkid '1234574' --datadir '~/Library/Ethereum' console
```

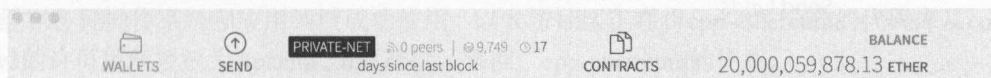
```
I0613 11:27:09.694551 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to
/Users/chenyufeng/Library/Ethereum/geth/chaindata
I0613 11:27:09.708544 ethdb/database.go:176] closed
db:/Users/chenyufeng/Library/Ethereum/geth/chaindata
I0613 11:27:09.709019 node/node.go:175] instance:
Geth/v1.5.2-stable-c8695209/darwin/go1.7.3
I0613 11:27:09.709050 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to
/Users/chenyufeng/Library/Ethereum/geth/chaindata
I0613 11:27:09.761401 eth/backend.go:193] Protocol Versions: [63 62], Network Id: 1234574
I0613 11:27:09.763172 core/blockchain.go:214] Last header: #10939 [c6a1472d...] TD=18506474242
I0613 11:27:09.763196 core/blockchain.go:215] Last block: #10939 [c6a1472d...] TD=18506474242
I0613 11:27:09.763204 core/blockchain.go:216] Fast block: #10939 [c6a1472d...] TD=18506474242
I0613 11:27:09.764489 p2p/server.go:336] Starting Server
I0613 11:27:09.764653 p2p/server.go:604] Listening on [::]:30303
I0613 11:27:09.766522 node/node.go:340] IPC endpoint opened:
/Users/chenyufeng/Library/Ethereum/geth.ipc
I0613 11:27:09.766808 node/node.go:410] HTTP endpoint opened: http://0.0.0.0:8545
I0613 11:27:10.823407 cmd/geth/accountcmd.go:200] Unlocked account
90c2323cdeff75fd82e65ac496fc45eafadf4563
Welcome to the Geth JavaScript console!
```

```
instance: Geth/v1.5.2-stable-c8695209/darwin/go1.7.3
coinbase: 0x90c2323cdeff75fd82e65ac496fc45eafadf4563
at block: 10939 (Thu, 08 Jun 2017 23:52:17 CST)
datadir: /Users/chenyufeng/Library/Ethereum
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0
personal:1.0 rpc:1.0 txpool:1.0 web3:1.0
>
```

2. 以太坊钱包

以太坊钱包（Ethereum Wallet）是一个以太坊账户管理的独立应用，它是开源的，任何人都可以提交代码改进项目，同时可以离线管理账户，包括账户的创建、备份、导入、更新等。以太坊钱包最重要的功能是可以进行以太坊交易。目前官方下载地址为<https://github.com/ethereum/>

mist/releases。图2.1所示为以太坊钱包主界面。安装完成以后，以太坊钱包会同步全网的信息，同步完成以后可以进行创建账户（见图2.2）、设置密码、转账（见图2.3）等操作。



Accounts Overview

ACCOUNTS

Accounts are password protected keys that can hold ether, secure ethereum-based tokens or coins and control contracts. Accounts can't display incoming transactions.



图2.1 以太坊钱包界面

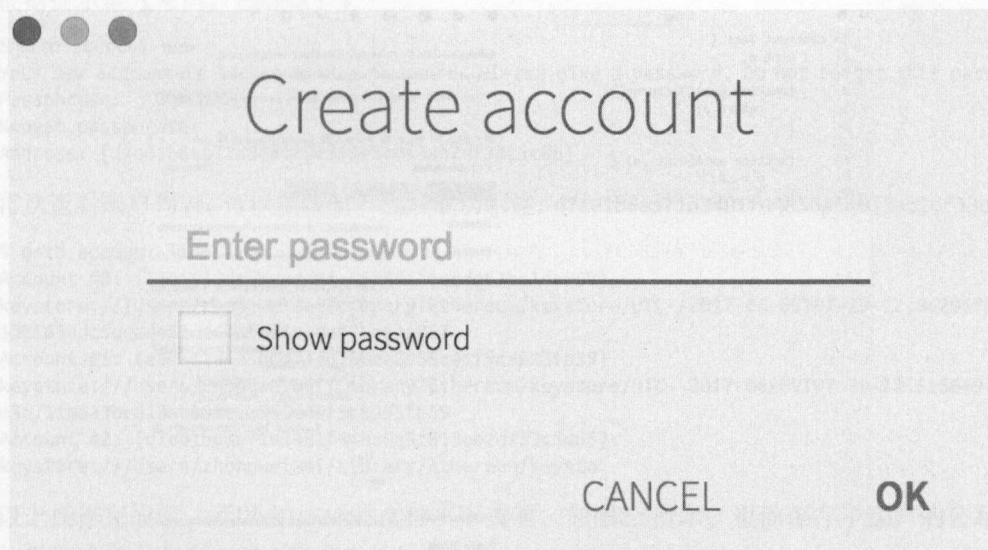


图2.2 以太坊钱包创建账户

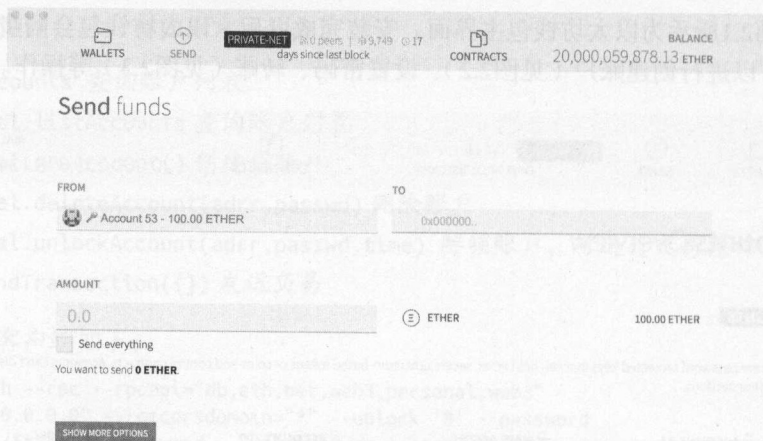


图2.3 以太坊钱包发送以太币

3. browser-solidity (Remix)

browser-solidity是一个在线浏览器编译器，是使用C++开发的，又叫Remix。官网地址为<https://ethereum.github.io/browser-solidity/>。该客户端不需要安装，在各个操作系统中使用都一样，适合初学者入门，可以进行合约的编译部署和测试。在智能合约开发中，browser-solidity是最常用的工具之一。

browser-solidity主界面如图2.4所示。

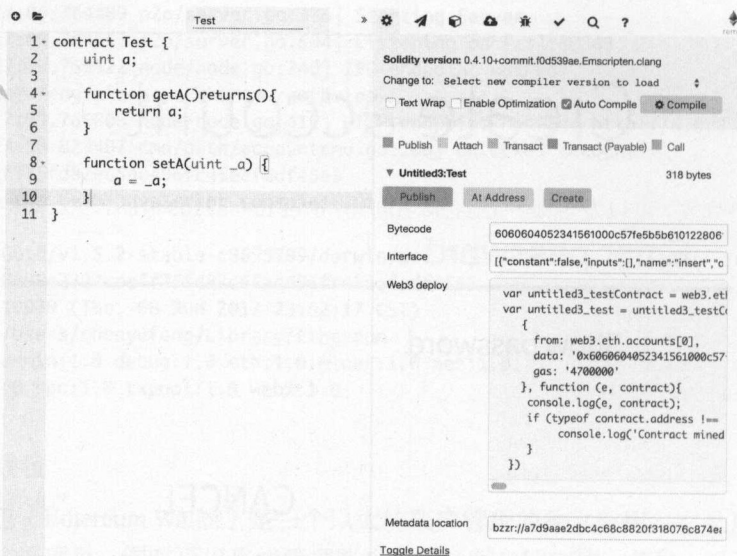


图2.4 browser-solidity主界面

4. cpp-ethereum

这是以太坊客户端的C++实现，流行程度仅次于geth。cpp-ethereum的代码非常便于移植，并且已经在各种操作系统和应用领域成功使用。以太坊社区正在将cpp-ethereum的代码库从copyleft GPLv3的许可重新授权为Apache 2.0的授权，来推广cpp-ethereum的使用。

2.1.4 以太坊账户管理

账户是操作以太坊的一把钥匙。外部账户之间可以进行交易，每个账户由公私钥定义，以地址为索引。地址由公钥的最后20字节衍生而来，并把钥匙对编码在JSON中成为私钥文件。账户私钥用来对发送的交易进行加密。所有创建的账户信息都存放在以太坊安装目录的keystore下。keystore文件可以进行备份，用户只有同时拥有钥匙文件和密码，才能进行交易。注意，当私钥丢失时，也就意味着这个节点的账户丢失了。下面介绍账户的管理。

1. 使用geth命令行

查看当前的3个账户：

```
$ geth account list
Account #0: {1301639dc5dcade8fe4672faf1acda67bc14a057}
keystore:///Users/zhongweiwei/Library/Ethereum/keystore/UTC--2017-06-09T07-29-12.862968835Z--1301639dc5dcade8fe4672faf1acda67bc14a057
Account #1: {e3e711a6a3bcd13eb108ec2955c41f9c8995fb59}
keystore:///Users/zhongweiwei/Library/Ethereum/keystore/UTC--2017-06-09T07-30-12.515649743Z--e3e711a6a3bcd13eb108ec2955c41f9c8995fb59
```

根据提示输入密码，创建账户：

```
$ geth account new
Your new account is locked with a password. Please give a password. Do not forget this password.
Passphrase:
Repeat passphrase:
Address: {d7e01b6aa71d3481f64856a5c013ab2df33c1c86}
```

再次查看账户列表，可以发现账户已经成功创建：d7e01b6aa71d3481f64856a5c013ab2df33c1c86。

```
$ geth account list
Account #0: {1301639dc5dcade8fe4672faf1acda67bc14a057}
keystore:///Users/zhongweiwei/Library/Ethereum/keystore/UTC--2017-06-09T07-29-12.862968835Z--1301639dc5dcade8fe4672faf1acda67bc14a057
Account #1: {e3e711a6a3bcd13eb108ec2955c41f9c8995fb59}
keystore:///Users/zhongweiwei/Library/Ethereum/keystore/UTC--2017-06-09T07-30-12.515649743Z--e3e711a6a3bcd13eb108ec2955c41f9c8995fb59
Account #2: {d7e01b6aa71d3481f64856a5c013ab2df33c1c86}
keystore:///Users/zhongweiwei/Library/Ethereum/keysto
```

以上操作完成后，可以去keystore文件夹下查看，如图2.5所示，里面保存了账户的公私钥文件。如果创建多个账户，就会有多个地址。

```

[ ] UTC--2017-06-09T07-29-12.862968835Z--1301639dc5dcade8fe4672faf1acda67bc14a057
[ ] UTC--2017-06-09T07-30-12.515649743Z--e3e711a6a3bcd13eb108ec2955c41f9c8995fb59
[ ] UTC--2017-06-09T07-33-36.585818367Z--d7e01b6aa71d3481f64856a5c013ab2df33c1c86

```

图2.5 keystore下查看公私钥文件

2. 使用geth控制台

进入geth控制台也可以执行和geth命令行相同的操作。首先进入geth控制台：

```

$ geth console 2>> log_file_output
Welcome to the Geth JavaScript console!

instance: Geth/v1.6.0-stable-facc47cb/darwin-amd64/go1.8.1
coinbase: 0x1301639dc5dcade8fe4672faf1acda67bc14a057
at block: 0 (Thu, 01 Jan 1970 08:00:00 CST)
datadir: /Users/zhongweiwei/Library/Ethereum
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

```

根据提示输入密码，创建账户：

```

$ personal.newAccount()
Passphrase:
Repeat passphrase:
"0xfbcc605066317fd4f6bdf33d062d85513e3532c6"

```

查看所有账户，可以发现账户已经成功创建：

```

$ eth.accounts
["0x1301639dc5dcade8fe4672faf1acda67bc14a057", "0xe3e711a6a3bcd13eb108ec2955c41f9c8995fb59",
"0xd7e01b6aa71d3481f64856a5c013ab2df33c1c86", "0xfbcc605066317fd4f6bdf33d062d85513e3532c6"]

```

3. 使用以太坊钱包

以太坊钱包是可视化的客户端，可以非常方便地进行账户管理。以太坊钱包可以自动读取或写入keystore文件。

查看所有的以太坊账户界面，如图2.6所示。

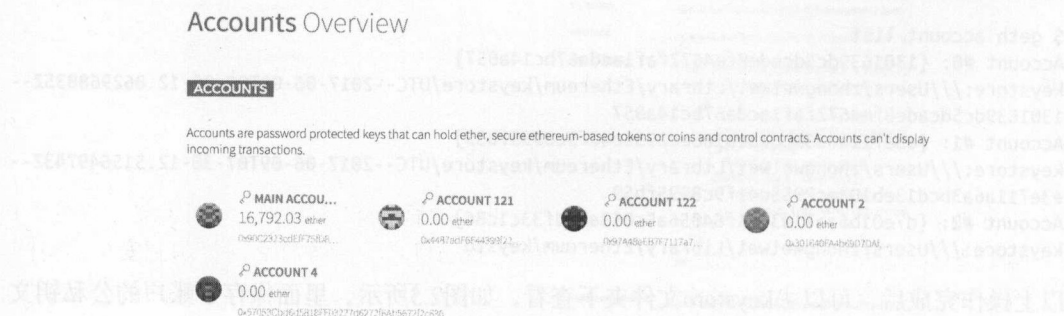


图2.6 以太坊钱包查看所有账户

创建一个以太坊账户，输入密码即可，如图2.7所示。

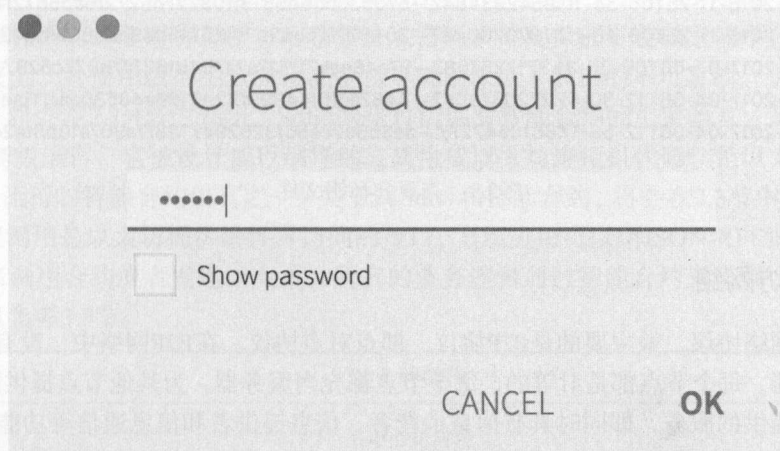


图2.7 以太坊钱包创建账户

再次查看当前所有的账户，de3b6c2ce56f3262947188f7a907a10b8fe24664已经成功创建。通过查看keystore目录，可以看到用户的账户文件已经成功写入，如图2.8所示。

Accounts Overview

ACCOUNTS

Accounts are password protected keys that can hold ether, secure ethereum-based tokens or coins and control contracts. Accounts can't display incoming transactions.

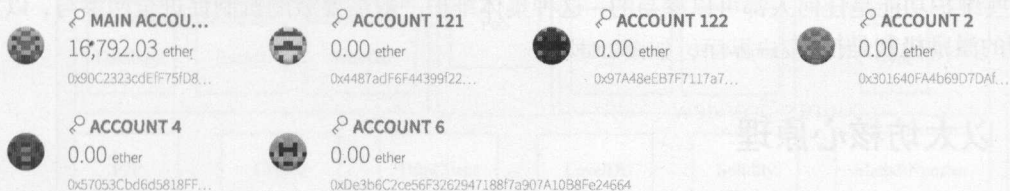



图2.8 以太坊钱包成功创建账户

在以太坊客户端的安装目录下有一个keystore目录，所有的账户文件都存储于此。对这个目录的复制是安全的。所以常规的账户备份策略是把keystore目录下的所有文件复制到安全的空间。当需要恢复账户时，将账户文件再次复制到keystore文件夹下即可。经过以上步骤后，当前的所有以太坊账户如图2.9所示。



```
UTC--2016-11-25T05-21-28.600835727Z--90c2323cdeff75fd82e65ac496fc45eafadf4563
UTC--2016-11-25T06-34-10.954293484Z--57053cbd6d5818ff03227d6272f6ab5672f2c636
UTC--2016-11-25T06-46-03.797876046Z--301640fa4b69d7daf07a5d63a85808fd8739c0f2
UTC--2017-04-06T09-25-31.371723768Z--97a48eeb7f7117a7a7b4b1523f7b072c528734ac
UTC--2017-04-06T12-30-47.272037804Z--4487adf6f44399f22d6169a46530aaf111ae109e
UTC--2017-04-06T12-55-17.851054727Z--de3b6c2ce56f3262947188f7a907a10b8fe24664
```

图2.9 查看公私钥文件

2.1.5 以太坊网络

区块链的网络协议，最主要的是P2P协议，即点对点协议。在P2P网络中，没有中心服务器，没有中心路由器，每个节点都是对等的，每个节点都充当服务器，为其他节点提供服务，同时也享用其他节点提供的服务，即同时具有信息消费者、信息提供者 and 信息通信等功能。P2P网络体系是去中心、去信任和集体协作的网络体系，基于此，以太坊区块链有如下特点。

(1) 去信任化。因为以太坊中没有中心节点，它们之间建立信任关系是通过成熟的密码学来保障交易的可靠性，所以参与整个系统的所有节点之间进行的数据交换是无需信任的。

(2) 去中心化。P2P网络是分布式的，交易的各方节点之间的信任关系是完全不需要借助第三方节点的，解决了中心节点的强依赖问题。

(3) 数据可靠。在整个以太坊中，每个参与的节点都有一份区块链数据的备份，所以单个节点对数据的修改是无效的，并且这个节点的错误数据会被其他节点共同修正，除非能同时控制51%的节点才能恶意修改数据，因此，参与系统的节点越多，计算能力越强，该系统的数据安全性越高。

(4) 集体协作。系统中的所有区块是整个系统中具有维护功能的节点共同维护的，在公有链中，这些维护功能是任何人都可以参与的，这种集体维护一般需要激励机制促进全网参与，以太坊采用的激励机制是挖矿。

2.2 以太坊核心原理

以太坊作为区块链的一个较为成熟的平台，以其安全可靠和易用性被很多开发者和公司所信任。以太坊的整体架构如图2.10所示。

以太坊最底层主要包括P2P协议，这是一种不存在中心服务器、两个节点之间直接进行网络通信的协议，只有基于P2P，区块链才能提供去中心化的服务。共识算法是区块链平台的核心组成部分，是不同节点之间达成一致性的算法和策略，目前以太坊最重要的两种共识算法是PoW和PoS，会在2.2.1节中介绍。EVM即以太坊虚拟机，是去中心化应用运行的容器，智能合约被编译成字节码后可以运行在EVM中，关于EVM的详细执行流程，将会在2.2.2节中进行概述。

LevelDB是以太坊底层的数据库，是由谷歌实现的非常高效的键值（key-value）数据库，目前很多基于企业级的区块链平台底层也是使用LevelDB进行存储的，关于以太坊的数据存储，可以参考2.2.3节。多种不同的非对称加密、哈希算法从密码学角度保证了在以太坊平台上的账户安全和交易信息安全，并使用数字签名和验证签名等机制保证了数据的不可篡改，关于以太坊涉及的加密算法，将会在2.2.4节中介绍。Solidity是目前编写智能合约的主要语言，是一种语法类似JavaScript的高级语言，它被设计成以编译的方式生成以太坊虚拟机代码，是以太坊推荐的旗舰语言，也是最流行的智能合约语言之一，关于Solidity和智能合约，将会在2.3节中进行深入讲解。RPC远程过程调用是以太坊提供给外界访问的接口，上层应用可以用JSON-RPC的方式和以太坊进行交互，来调用合约或者发送以太币，所有的业务逻辑通过智能合约来实现，关于以太坊编程接口，可以参考3.3节。

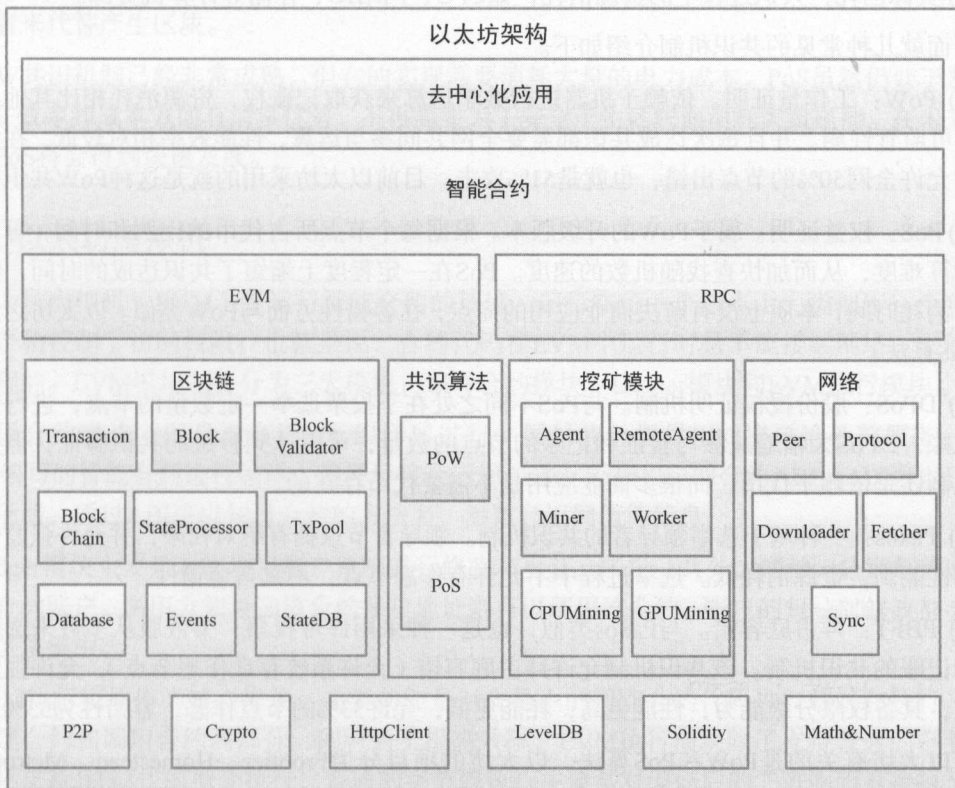


图2.10 以太坊架构

2.2.1 以太坊共识机制

共识机制是多个节点之间达成一致性的一种数学算法。在区块链中，共识机制的作用显得尤为重要。由于区块链中每个节点都是相互独立的，而每一个节点都存有分布式账本的完全备份，

如何对这些账本数据进行一致性验证就是共识机制需要考虑的问题。换句话说,共识机制就是在不同节点之间建立信任,获取权益的数学算法。它允许关联机器连接起来进行工作,并在某些成员失效的情况下仍能正常运行。

常见的共识机制有工作量证明算法、权益证明、股份授权证明和拜占庭容错等,基于不同的应用场景和共识机制等特性,可以通过以下维度来评价共识机制的优劣。

- (1) 合规监管:是否支持超级权限节点对全网节点、数据进行监管。
- (2) 性能效率:交易达成共识并且确认的效率。
- (3) 容错性:防攻击、防欺诈的能力。
- (4) 资源消耗:共识过程中的资源消耗,如CPU、网络IO、存储等计算机资源。

下面就几种常见的共识机制介绍如下。

(1) PoW:工作量证明。依赖于机器进行数学运算来获取记账权,资源消耗相比其他共识机制高,可监管性弱。并且每次达成共识都需要全网共同参与运算,性能效率相对较低,在容错性方面,允许全网50%的节点出错,也就是51%攻击。目前以太坊采用的就是这种PoW共识机制。

(2) PoS:权益证明。属于PoW的升级版。根据每个节点所占代币的比例和时间,等比例地降低计算难度,从而加快查找随机数的速度。PoS在一定程度上缩短了共识达成的时间,但是还是需要消耗时间,本质上没有解决商业应用的痛点,在容错性方面与PoW类似。以太坊之后将会转为PoS算法。

(3) DPoS:股份授权证明机制。与PoS不同之处在于投票选举一定数量的节点,进行代理验证和记账。DPoS大幅缩减参与验证和记账的节点的数量,可以达到秒级的共识验证,但是整个共识机制还是依赖于代币,而很多商业应用是不需要代币存在的。

(4) Paxos:一种基于选举领导者的共识机制。领导者节点拥有绝对权限,并允许强监管节点参与。性能高,资源消耗低。选举过程中不允许有作恶节点,不具备容错性。

(5) PBFT:拜占庭容错。与Paxos类似,也是一种采用许可投票、少数服从多数来选举领导者进行记账的共识机制。该共识机制允许拜占庭容错(允许系统存在作恶节点),允许强监管节点参与,具备权限分级能力,性能更高,耗能更低,允许33%的节点作恶,容错性为33%。

与以太坊有关的是PoW和PoS算法。以太坊的项目分为Frontier、Homestead、Metropolis和Serenity四个阶段。在前面三个阶段,以太坊共识算法采用的是PoW,在第四个阶段会转移到PoS。

工作量证明是若一个节点付出了足够的算力,那么区块链认为这个节点发现的区块是有效的。比特币是最先采用PoW记账方式的区块链项目,矿工通过计算机算力来争夺比特币的记账权,每当一个比特币区块产生时,系统会为最先成功记账的矿工给予一定数量的奖励。工作量证明的目的是使下一个区块的创建变得困难,从而阻止攻击者恶意生成区块链。PoW是对每一个区块进行SHA256哈希运算,将得到的哈希值看作长度为256位的不可预测的数值,要保证该数值小于不

断动态调整的目标数值。假如当前目标数值为 2^{192} ，就意味着平均需要尝试 2^{64} 次才能生成有效的区块。一般来说，比特币网络每隔2016个区块重新设置目标数值，保证平均每十分钟生成一个区块。而以太坊产生区块的时间大约为14秒。

而权益证明PoS是一种公有区块链中的共识算法，用来替换工作量证明。可以被描述成一种虚拟挖矿，依赖于区块链自身的代币。在PoS中，用户通过购买等价值的代币当作押金放入PoS机制中，就有机会产生新块而得到奖励。大致过程可以描述为，存在一个持有代币的用户集合，他们把手中的代币放入PoS机制中，这些用户就变成了验证者。假设在区块链最新的一个区块中，PoS算法在这些验证者中根据权重随机挑选，权重依据验证者投入的代币多少，比如一个用户押金为1000代币，另一个投入100代币，则前一个用户选中的概率是后一个用户的10倍。选中用户后，给他们权利产生下一个区块，如果一定时间内，这个选中的验证者没有产生区块，则选第二个验证者来代替产生区块。

PoW共识机制已经非常成熟，但它的实现需要消耗大量的电力成本，PoS虽然仍处于发展阶段，但在效率和成本方面有诸多优势，不再需要为了安全产生区块而消耗大量电能。在未来一段时间，PoS将会得到快速发展。

2.2.2 以太坊虚拟机

以太坊虚拟机（EVM）是运行智能合约的环境，运行在每一个节点上，类似于一个独立的沙盒，严格控制了访问权限；也就是说，合约代码在EVM中运行时是不能接触网络、文件或者其他进程的。EVM模块主要分为三大模块：编译合约模块、Ledger模块和EVM执行模块。

编译合约模块主要是对底层Solc编译器进行一层封装，提供RPC接口给外部服务，对用Solidity编写的智能合约进行编译。编译后将会返回二进制码和相应的合约ABI，ABI可以理解为用户的手册，通过ABI可以知道合约的方法名、参数、返回值等信息。

Ledger模块主要是对区块链账户系统进行修改和更新，账户一共分为两种，分别是普通账户和智能合约账户，调用方如果知道合约账户地址则可以调用该合约，账户的每一次修改都会被持久化到区块链中。

EVM执行模块作为核心模块，主要功能是对交易中的智能合约代码进行解析和执行，一般分为创建合约和调用合约两部分。同时为了提高效率，EVM执行模块除了支持普通的字节码执行外，还支持JIT模式的指令执行，普通的字节码执行主要是对编译后的二进制码直接执行其指令，而JIT模式会对执行过程中的指令进行优化，如把连续的push指令打包成一个切片，方便程序高效执行。EVM执行交易的流程如图2.11所示。

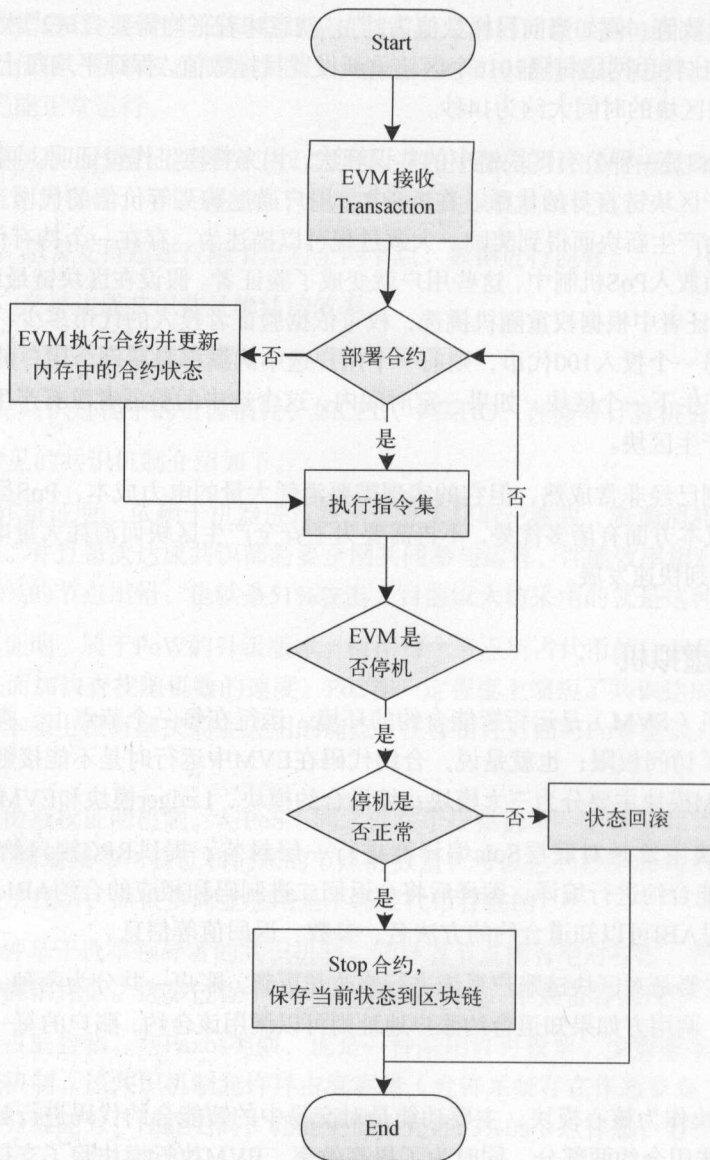


图2.11 EVM模块执行流程

(1) EVM接收到Transaction信息, 然后判断Transaction类型是部署合约还是执行合约。如果是部署合约, 执行指令集, 来存储合约地址和编译后的代码; 如果是执行合约或是调用合约, 则使用EVM来执行输入指令集。

(2) 执行上一条指令集之后, 判断EVM是否停机, 如果停机则判断是否正常停机, 正常停机则更新合约状态到区块链, 否则回滚合约状态。如果不停机则回到上一步(1)进行判断。

(3) 执行完的合约会返回一个执行结果，EVM会将结果存储在Receipt回执中，调用者可以通过Transaction的哈希来查询结果。

2.2.3 以太坊数据存储

以太坊中存储数据的结构是Merkle Patricia Tree，即默克尔-帕特里夏树。Merkle树是由计算机科学家Ralph Merkle提出的，在比特币网络中使用了这种数据结构来进行数据的正确性验证。而以太坊结合了Merkle树和Patricia树并进行了优化。

在比特币网络中，Merkle树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹。Merkle树是自底向上构建的，在图2.12中，首先将L1~L4这4个单元数据哈希化，然后将哈希值存储至相应的叶子节点中，这些节点分别是Hash0-0、Hash0-1、Hash1-0、Hash1-1。

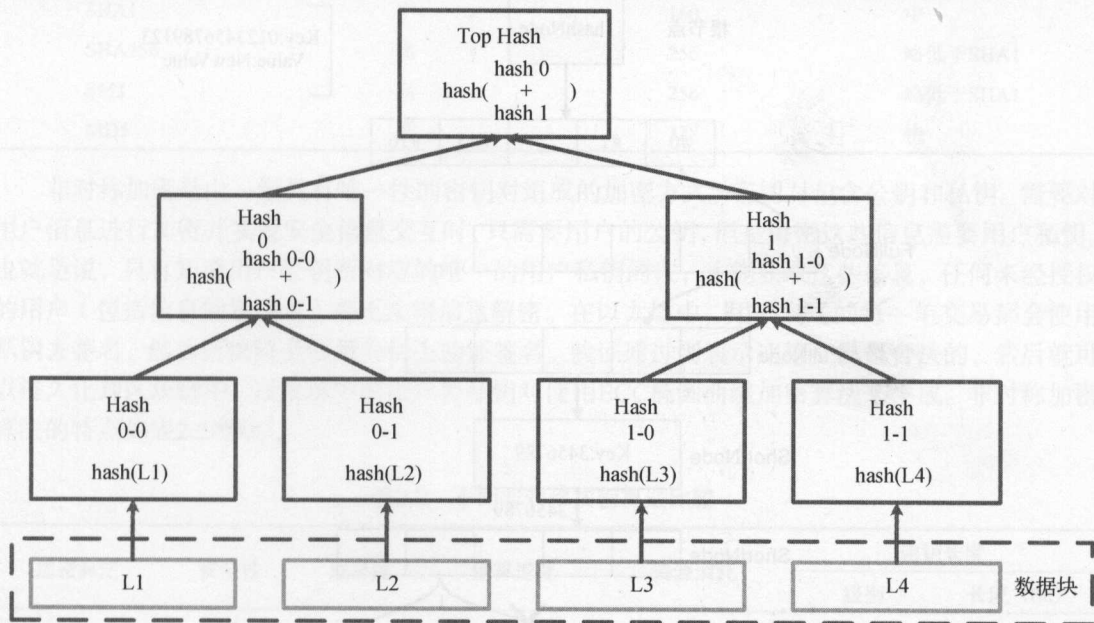


图2.12 Merkle树结构

将相邻两个节点的哈希值合并成一个字符串，然后计算这个字符串的哈希，得到的就是这两个节点的父节点的哈希值。（如果最底层的哈希总数是单数，那到最后必然出现一个单身哈希，这种情况就直接对它进行哈希运算，得到的哈希值就是其父节点的哈希值。）

循环重复上述计算过程，计算得到最后一个节点的哈希值，将该节点的哈希值作为整棵树的Merkle根。若两棵树的Merkle根一致，则这两棵树的结构及每个节点的内容必然相同。

Merkle树的一个明显优势是验证过程中可以快速定位“不正确”数据的位置。由于原始数据

的一点差异就会造成计算所得的哈希都不正确，因此比较两棵Merkle树，很容易找到这条路径，从而定位“非正确”节点的位置。

由于Merkle树在比特币中的应用比较单一，尽管它可以证明包含的交易，但是它不能进行涉及当前状态的证明（如数字资产的持有、名称注册、金融合约的状态等），且Merkle树在以太坊中主要的操作就是构建树，而对于树节点内容的更改、插入等操作十分不便。为了扩展这些操作，以太坊对Merkle树进行了修改，并融合了Patricia树。

Patricia树会存储每个账户的状态。这个树的建立是通过从每个节点开始，将节点分成多达16个组，然后哈希每个组，再对哈希结果继续哈希，直到整棵树有一个最后的“根哈希”。Patricia树很容易进行更新、添加或者删除树节点，以及生成新的根哈希。使用Patricia树进行更新的示例如图2.13所示。

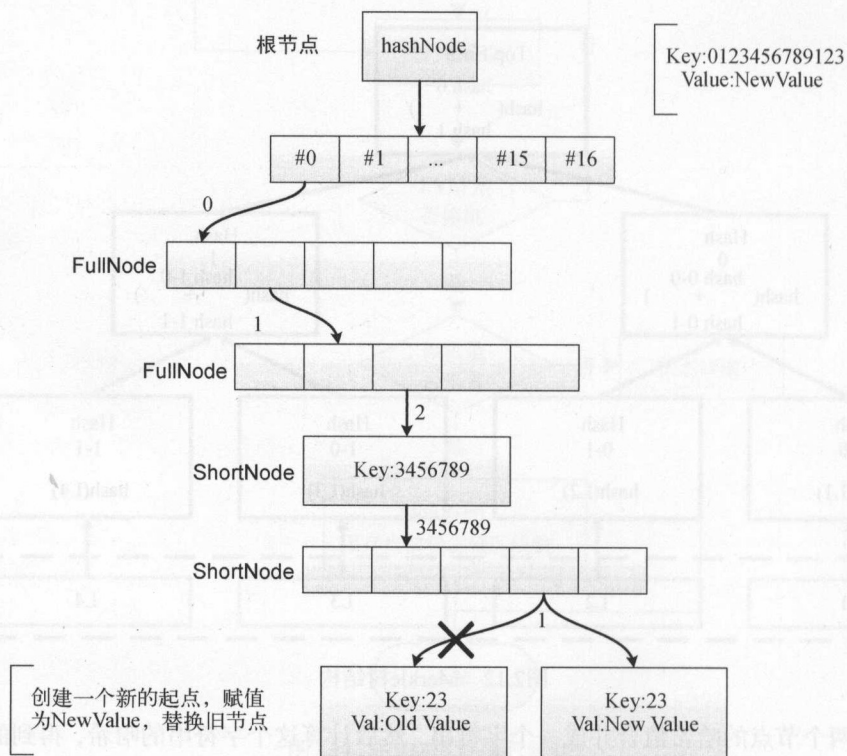


图2.13 Patricia树更新示例

(1) 将根节点传入作为当前处理节点，传入目标节点的Key作为路径。

(2) 传入新的Value值，若Value值为空，则找到该节点并删除；反之，创建一个新节点替换旧节点。

2.2.4 以太坊加密算法

以太坊使用了多种加密算法，最主要的是哈希算法和非对称加密算法。

哈希算法又叫散列算法，在以太坊中用来快速验证用户的身份，其原理是将一段信息或文本转化成有一个固定长度的字符串（摘要）。哈希算法的特点是如果某两段信息完全相同，那么，最终加密后的字符串也完全相同；如果两段信息不完全相同，即使只有一个字符不同，那么，最终的字符串都会十分杂乱和随机，并且两个字符串之间没有任何关联性。目前以太坊区块链主要使用SHA256来进行挖矿运算。不同的哈希算法的特点如表2.1所示。

表2.1 不同哈希算法比较

算法类别	安全性	输出大小（bit）	运算速度
SHA1	中	160	中
SHA256	高	256	略低于SHA1
SM3	高	256	略低于SHA1
MD5	低	128	快

非对称加密是由一组具有唯一性的密钥对组成的加密方式，密钥对包含公钥和私钥。需要对用户信息进行加密并实现安全信息交互时，只需要用户的公钥，但是解密这些信息需要用户私钥。也就是说，只有知道用户公钥所对应的唯一的用户私钥的人，才能获取这些信息，任何未经授权的用户（包括信息的发送者）都无法将信息解密。在以太坊中，用户发送的每一笔交易都会使用私钥去签名，然后区块链会使用公钥去验证签名。验证通过则表示这笔交易是合法的，然后就可以持久化到区块链中。以太坊中的用户公私钥对使用ECC椭圆曲线加密算法来生成。非对称加密算法的特点如表2.2所示。

表2.2 不同非对称加密算法比较

加密算法	安全性	成熟度	运算速度	资源消耗	密钥长度	
					级别	长度（bit）
RSA	低	高	慢	高	80	1024
					112	2048
ECC	高	高	中	中	80	160
					112	224
SM2	高	高	中	中	80	160
					112	224

2.3 以太坊智能合约

本节将详细介绍如何使用Solidity编写智能合约，首先会对Solidity做一个简要的介绍，然后

详细地介绍Solidity的基本语法，如数据类型、状态变量、函数调用等，同时Solidity作为一门面向对象的语言，也会有相应的面向对象的性质。编写完智能合约后，最重要的是智能合约的测试，本节会用不同的测试框架对合约进行测试，以此来验证合约是否符合预期。最后会对一个较为复杂的智能合约进行实例分析，使读者加深对智能合约的了解。

2.3.1 智能合约与 Solidity 简介

什么是智能合约？合约是代码（逻辑描述）和数据（状态表示）的集合，存储在以太坊区块链的特定地址。合约账户能够在彼此之间传递信息，进行图灵完备的运算，编译成EVM字节代码（以太坊特有的二进制格式），并运行在区块链上。换句话说，智能合约是运行在区块链上的模块化、可重用、自动执行的脚本。合约部署的时候将编译器编译得到的字节码存储在区块链上，对应会有一个存储地址。当预定的条件发生时，就会发送一笔交易到该合约地址，全网节点都会执行合约脚本编译生成的操作码，最后将执行结果写入区块链。所以，可以把智能合约理解为在区块链上执行操作的所有业务逻辑代码。

智能合约一个重要的特点是图灵完备。图灵完备是指一个能计算出每个图灵可计算函数的计算系统，图灵完备使脚本系统有能力解决所有的可计算问题。智能合约是图灵完备的，即可以实现图灵机所能做到的所有事情。通俗来讲，一般编程语言可以做到的所有逻辑操作，在智能合约中都可以实现。

智能合约另一个重要的特点是沙箱隔离。对I/O、网络操作、访问其他进程等进行了限制，实际上是完全隔离的。所以，目前实现的智能合约无法进行文件的读取和写入，也无法实现网络资源的访问或直接提供网络服务。智能合约只能在部署到区块链平台上以后，才能使用区块链平台提供的接口进行合约数据的访问，即访问智能合约中的数据和方法。该特性提高了智能合约执行的安全性。

目前可以使用Solidity、Serpent、LLL、Mutan这些语言来编写智能合约，但是使用最广泛、最受欢迎的还是Solidity。

Solidity是一种语法类似于JavaScript的高级面向对象语言，也是一门静态类型语言，被设计用来编写智能合约，并运行在以太坊虚拟机之上。Solidity支持继承、库以及复杂的自定义类型。文件扩展名为.sol，是一种真正意义上的运行在网络上的去中心化合约。目前Solidity有在线的实时编译器，方便开发者使用。此外还支持多种标准的库函数，并具有以下特性。

- ❑ 以太坊底层是基于账户的，所以有一个特殊的Address类型，用于定位用户，定位合约，以及定位合约的代码（合约本身也是一个账户）。
- ❑ 由于合约内嵌框架是支持支付的，所以提供了一些关键字，如payable，可以在语言层面直接支付。
- ❑ 使用网络上的区块链存储，数据的每一个状态都可以永久存储，所以需要确定变量使用

内存还是区块链。

- 一旦出现异常，所有的执行都将被回滚，这主要是为了保证合约执行的原子性，以避免中间状态出现数据不一致。

目前使用Solidity可以很容易地创建用于投票、众筹、封闭拍卖、多重签名钱包等应用场合的合约。以下是Solidity官方介绍的一些开发工具。

(1) Browser-Based Compiler^①

这是Solidity官方强烈推荐的基于浏览器的在线IDE，也称为Remix。Browser-Based Compiler已经集成了编译器和Solidity运行环境，并且不需要任何服务端组件。

(2) Visual Studio Extension^②

一款Visual Studio上的Solidity插件，提供了Solidity编译器。

(3) Package for SublimeText-Solidity language syntax^③

一款用于SublimeText编辑器的Solidity语法高亮工具。

(4) Atom Ethereum interface^④

用于Atom编辑器的插件，提供Solidity语法高亮、编译和实时运行环境（需要以太坊节点）。

(5) Emacs Solidity^⑤

用于Emacs编辑器的插件，用于语法高亮、编译和错误报告。

(6) Vim Solidity^⑥

用于Vim编辑器上的插件并提供语法高亮。

(7) Mix IDE^⑦

一款基于Qt的合约集成开发环境，可用于设计、调试、测试Solidity写的智能合约，在本书第3章有详细的介绍。但是目前已经不再更新维护，开发中使用较少。

开发者可以根据项目需求和自己的喜好，使用上述的一款或多款工具进行基于Solidity的智能合约的开发。

① <https://ethereum.github.io/browser-solidity/>。

② <https://marketplace.visualstudio.com/items?itemName=ConsenSys.Solidity>。

③ <https://packagecontrol.io/packages/Ethereum>。

④ <https://github.com/gmtDevs/atom-ethereum-interface>。

⑤ <https://github.com/ethereum/emacs-solidity/>。

⑥ <https://github.com/tomlion/vim-solidity/>。

⑦ <https://github.com/ethereum/mix/>。

2.3.2 智能合约的编写与部署

本节将从一个简单的HelloWorld智能合约程序入手，学习智能合约的数据类型、方法调用、事件等重要组成部分，以及如何把编写完成的智能合约部署到以太坊环境中，使以太坊自动执行该合约中的逻辑。下面是这个简单的Solidity-HelloWorld案例。

1. Solidity-HelloWorld

学习每一门语言都是从HelloWorld程序开始，学习Solidity也不例外。Solidity在以太坊环境内操作，没有明显的“输出”字符串的方式，这里就用日志记录事件来把字符串放进区块链中。HelloWorld的合约实现如下。每次合约被创建时，都会调用构造函数，合约都会在区块链创建一个日志入口，打印HelloWorld参数。

```
contract HelloWorld {
    event Print(string out);
    function HelloWorld() {
        //发送Print事件
        Print("Hello World!");
    }
}
```

合约在在线开发工具Remix中的执行结果如图2.14所示，成功从事件中接收到“Hello World!”。

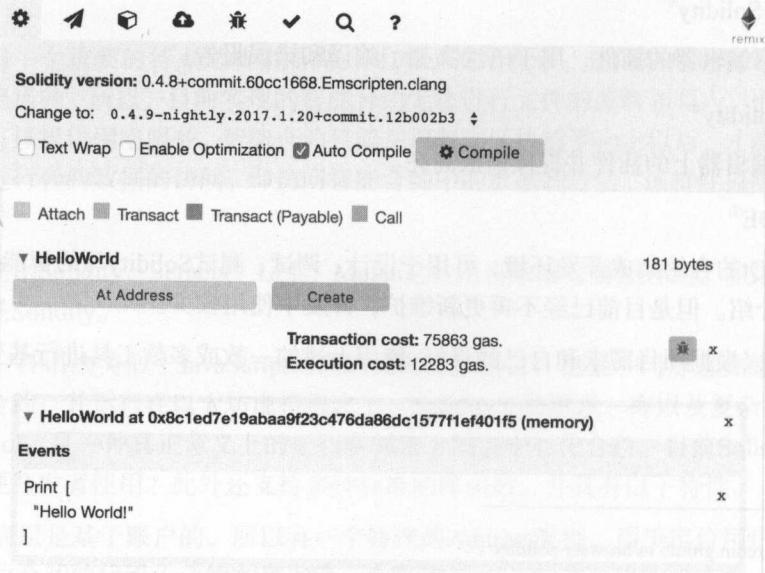


图2.14 Remix执行HelloWorld

2. 文件布局与合约结构

Solidity的合约和面向对象语言中的类的定义相似，每个合约包括了状态变量、函数、函数修

饰符、事件、结构体类型、枚举类型和注释。另外，合约也可以从其他合约中继承。合约不同组成部分的作用如下。

(1) 编译版本声明

一般在线编译器会要求指定Solidity编译器的版本号，否则会给出警告。目前一般要高于0.4版本才可以编译，声明语句如下：

```
pragma solidity ^0.4.0;
```

(2) 状态变量

在合约中用于永久存储变量的值，使用uint、int、bytes32、string、address等数据类型来修饰一个状态变量。

(3) 函数

函数是合约中一段可执行的代码块，使用function关键字进行修饰，函数可以有零个或多个输入参数，可以有零个或多个返回值。

(4) 函数修饰符

可以在声明函数时补充函数的语义，如使用constant、internal、external、public、private来修饰一个函数。不同修饰符的函数有不同的访问权限。

(5) 事件

事件是和EVM日志的接口，使用event进行声明，可以有一个或多个返回值。可以在函数内被调用，并在JavaScript API中被捕获。

(6) 结构体

结构体是一组自定义的复合数据类型，可以包含多个不同数据类型的状态变量。一般对一个对象的描述使用结构体。

(7) 枚举

枚举是用来创建一组特定值的集合的类型，使用enum声明，Solidity对枚举的底层处理就是uint类型。

3. 数据类型与状态变量

Solidity是一种静态类型语言，每个变量在编译时都需要被定义（包括全局变量或局部变量）。Solidity也提供了一些基础数据类型可以组成复杂类型。有一种变量类型叫值类型，其变量总是要被赋值，作为函数参数或者在赋值中，复制出一个全新的值。值类型包括布尔、整型、地址、字节数组、有理数和整型、十六进制字面量、枚举类型、函数。

(1) 布尔类型

布尔类型只有真值或假值两种。操作符包括：！（逻辑非）、&&（逻辑与）、||（逻辑或）、==

(相等)、!= (不等)。常规的短路规则同样适用于||操作符和&&操作符：对于表达式 $f(x)||g(y)$ ，如果 $f(x)$ 为真，则 $g(y)$ 不需要计算，因为结果一定为真；对于表达式 $f(x)&&g(y)$ ，如果 $f(x)$ 为假，则 $g(y)$ 不需要计算，因为结果一定为假。布尔类型的使用基本同其他高级编程语言。

(2) 整型

int/uint分别是有符号和无符号的整数，其中包括uint8到uint256，步长为8（从8到256位的无符号整数）。uint和int分别是uint256和int256的别名。

使用到整型的操作符有比较操作符：<=、<、==、!=、>=、>；位操作符：&（按位与）、|（按位或）、^（按位异或）、~（按位取反）；算术运算符：+、-、一元-、一元+、*、/、%（取余数）。整型的使用如下所示。

```
pragma solidity ^0.4.0;
contract IntExample{
    uint value;
    //两个uint值相加
    function add(uint x, uint y) returns (uint){
        value = x + y;
        return value;
    }
    //两个uint值相除，规则与C语言相同
    function divide() returns (uint){
        uint x = 3;
        uint y = 2;
        value = x / y;
        return value;
    }
}
```

(3) 地址

一个以太坊地址有20个字节，地址变量用address修饰。address可以修饰一个合约地址或者账户地址。地址也可以用比较操作符操作。地址类型变量默认有两个非常重要的方法：账户余额balance和发送send。可以使用address(0)来表示一个空地址，即0x0，如下述代码所示。

```
pragma solidity ^0.4.0;
contract AddressExample{
    bool isSuccess;
    //发送以太坊
    function sendEth(address account, uint amount) returns (bool){
        if (account.balance >= amount ){
            isSuccess = account.send(amount);
        } else {
            //余额不足，发送失败
            isSuccess = false;
        }
        return isSuccess;
    }
}
```

其中, `account` 是一个合约地址, 当执行 `account.send()` 函数时, 合约对应的代码会触发执行, 如果执行失败, 这个发送操作就会回滚。 `send()` 方法执行有一些注意事项, 例如调用的递归深度不能超过1024; 如果gas不够, 执行会失败, 所以使用 `send()` 方法后需要根据返回值判断是否成功。

(4) 字节数组

`bytes1`、`bytes2`、`bytes3`、...、`bytes32` (`bytes1` 等同于 `byte`) 都是定长字节数组。字节数组同样也可以使用比较运算符和位运算符。默认用只读成员变量 `.length` 表示这个字节数组的长度。

`bytes` 和 `string` 是两种动态长度的字节数组。`bytes` 一般用来表示任意长度的字节数据, `string` 用来表示任意长度的UTF-8编码的字符串数据。如果长度可以确定, 尽量使用定长的, 如 `bytes1` 到 `bytes32`, 因为这样占用空间更小。

(5) 字符串

字符串字面量是指由单引号或双引号引起来的字符串。Solidity字符串与C语言不同, 不包含结束, "foo"这个字符串的大小仅包含3个字节。字符串本身就是一种不定长的字节数组, 可以隐式地转换为`bytes1`、...、`bytes32`。字符串字面量支持转义字符, 如 "\n"。

```
pragma solidity ^0.4.0;
contract StringExample{
    string name;
    function stringTest(){
        name = "Jack";
    }
}
```

(6) 数组

数组的大小可以在编译阶段确定 (静态数组), 也可以是不定长的 (动态数组)。对于存储器数组来说, 成员类型是任意的 (也可以是其他数组、映射或结构体)。在函数内部使用内存数组时, 若函数对外可见, 则数组元素不能是映射类型, 且只能是支持ABI的类型。一个定长数组, 如果长度是 k , 元素类型为 T , 那么可以表示为 $T[k]$, 而另一个变长的数组为 $T[]$ 。

类型 `bytes` 和 `string` 本质上是一种特殊的数组, `bytes` 类似于 `byte[]`; `string` 比较类似于 `bytes`, 但是暂时不提供长度 (`length`) 和按下标 (`index`) 方式的访问。相对来说, `bytes` 更节省空间。

数组有以下成员函数。

`length()`: 长度字段, 表示持有元素的数量。动态数组如果是存储器类型的, 则可以调整大小。调整的方式是通过 `.length` 改变对应的值。

`push()`: 动态类型数组和 `bytes` 都有一个 `push()` 函数, 用于添加新元素, 返回结果为新数组或 `bytes` 的长度。


```

pragma solidity ^0.4.0;
contract ArrayExample {
    uint[] uintArray;
    bytes b;
    function arrayPush() returns (uint){
        uint[3] memory a = [uint(1), 2, 3];
        uintArray = a;
        //添加到数组的最后一个位置
        return uintArray.push(4);
    }
    function bytesPush() returns (uint){
        b = new bytes(3);
        return b.push(4);
    }
}

```

(7) 映射/字典

映射(mapping)的定义方式为`_KeyType=>_KeyValue`。键的类型不能为mapping、struct、array等,值的类型则无限制。映射可以看作一个通过将所有键初始化得到的哈希表,值为对应到某个类型的默认值。但与哈希表不同的是,键仅存储了它的Keccak-256哈希,用来找到其相关的值。因此,映射查找速度特别快,也没有长度或排序的概念。

映射类型只能用于定义状态变量,或者在内部函数中作为存储器类型的引用。当然,映射的值类型也可以为映射,可以提供键值递归访问。以下例子中,使用`msg.sender`合约调用者作为键类型、`amount`作为值类型进行存储。

```

pragma solidity ^0.4.0;
contract MappingExample{
    mapping(address => uint) public balances;
    uint balance;
    function updateBalance(uint amount) returns (uint){
        //设置mapping中的值
        balances[msg.sender] = amount;
        //取出mapping中的值
        balance = balances[msg.sender];
        return balance;
    }
}

```

合约中的一个重要组成部分就是状态变量,变量值会永久存储在合约的存储空间中,也就是存储在这些状态变量中。可以使用上面的数据类型来声明不同类型的状态变量,存储不同类型的数据,如下所示。

```

pragma solidity ^0.4.0;
contract TypeExample{
    uint uintValue;
    bool boolValue;
    address addressValue;
    bytes32 bytes32Value;
    uint[] arrayValue;
}

```

```

string stringValue;
enum Direction{Left, Right}
mapping(address => uint) mappingValue;
}

```

合约中的变量（状态变量或局部变量）或者函数参数，都有一个重要的概念——数据存储位置。一般合约中，变量的默认存储位置是这样设置的：全局的状态变量和局部变量存在存储器（storage）中；函数参数和返回的参数存放在内存（memory）中。也可以使用关键字storage和memory来覆盖默认的设置。

数据存储因位置的不同导致变量赋值产生的结果也不同。在memory、storage和状态变量中相互赋值，总是会创建一个完全不相关的备份。给一个局部存储变量赋值会被赋予一个引用，即使这个值发生了变化，它还是会指向对应的状态变量。

4. 函数调用

函数是编程语言中最重要的部分，可以实现模块化编程，大大提高项目开发的效率和代码的可读性及复用性。Solidity中的函数类似于JavaScript中的函数。

(1) 内部调用

在当前的同一个合约中，函数可以直接进行内部调用，也可以进行递归调用。这些函数在EVM中被翻译为简单的跳转JUMP指令。

```

pragma solidity ^0.4.0;
contract FunctionExample {
    function g(uint a) returns (uint) {
        //调用同一个合约中的函数
        return f(a);
    }
    function f(uint a) returns (uint) {
        return a;
    }
}

```

(2) 外部函数调用

对于不同合约的函数则必须使用外部调用的方式，而不是直接通过JUMP调用。对于一个外部调用，所有函数的参数必须复制到内存中。

如果被调用的合约不存在，或调用的合约产生了异常，或者gas不足，均会造成函数调用发生异常，本次执行过程中对账本造成的影响将会回滚。

```

pragma solidity ^0.4.0;
contract User {
    function age() returns (uint) {
        return 25;
    }
}
contract FunctionExample {

```

```

User user;
function FunctionExample() {
    //获得合约地址
    address addr = new User();
    //强制类型转换, 获得合约对象
    user = User(addr);
}
function callUser() {
    //使用合约对象调用方法
    user.age();
}
}

```

(3) 命名参数调用

函数调用的参数, 可以通过指定名字的方式调用, 以任意的顺序传入参数, 方式是使用{}包含。但参数的类型和数量要与定义一致。

```

pragma solidity ^0.4.0;
contract FunctionExample {
    function add(uint first, uint second) returns (uint) {
        return first + second;
    }
    function callAdd() returns (uint){
        //传入参数的先后可以不一致
        return add({second: 2, first: 1});
    }
}

```

在某些函数中会遇到异常, 需要使用throw指令手动抛出一个异常。异常的作用是当前执行的调用被停止和状态的回滚(即所有状态和余额的变化都没有发生)。在Solidity中, 外部是不能捕获异常的。

(4) 函数可见性

函数可以被标记为external、public、internal、private, 其中默认为public。

external: 外部函数是合约接口的一部分, 所以可以从其他合约通过交易来发起调用, 一个外部函数f(), 不能通过f()的方式来直接调用, 而要通过this.f()外部访问的方式来调用。外部函数在接收大的数组数据时更加有效。

public: 公开函数是合约接口的一部分, 可以通过内部或者消息来进行调用。public类型是访问权限最大的修饰符。

internal: 内部函数只能通过内部访问(例如在当前合约中)调用, 或在继承的合约里调用。需要注意的是, 不能加前缀this。

private: 私有函数仅在当前合约中可以访问, 在继承的合约内不可访问。private是访问权限最小的修饰符。

需要注意的是，所有在合约内的东西对外部的观察者来说都是可见的，将某些东西标记为 `private`，仅仅阻止了其他合约进行访问和修改，但不能阻止其他人看到相关的信息。可见性标识符的定义位置一般在参数列表和返回关键字中间。

```
pragma solidity ^0.4.0;
contract FunctionExample {
    function func(uint a) private returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

(5) 常函数

函数也可以被声明为常量，这类函数需要保证不改变任何值，即不会导致区块链状态的变化。这类函数不会消耗gas，可以使用 `constant` 来修饰。如果不加 `constant`，也不会影响常函数的执行。如果把一个会导致区块链状态改变的方法声明为 `constant`，则该 `constant` 修饰符没有任何作用，同样不会影响常函数的执行。在编程过程中，如果确定一个函数为常函数，推荐加上 `constant` 修饰符，这是良好的编程习惯，有利于提高代码可读性。如下述代码所示，把函数修饰为 `constant`。

```
pragma solidity ^0.4.0;
contract FunctionExample {
    uint a;
    //一般把状态变量的get方法声明为constant
    function getA() constant returns (uint) {
        return a;
    }
}
```

5. 事件

事件让使用EVM日志内置功能变得容易，它可以调用监听了对应事件的DApp的JavaScript回调。当事件被调用时，会触发参数存储到交易的日志中，这些日志与合约地址关联，并合并到区块链中，只要区块可以访问就一直存在。日志和事件在合约内不可直接访问，即使是创建日志的合约也不行。在开发DApp中，事件可能需要经常使用。

事件最多有3个参数被设置为索引 (`indexed`)，让对应的参数值可以在以太坊虚拟机日志中被检索到。在用户调用接口时，输入对应的索引即可查找到对应的值。如果数组（包括 `string` 和 `bytes`）类型被标记为索引项，则查询到的值是该索引值的Keccak-256哈希值。所有未被索引的参数将被作为日志的一部分保存起来。事件合约代码示例如下：

```
pragma solidity ^0.4.0;
contract EventExample {
    //声明事件，可以有多个参数
    event SendBalance(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );
}
```

```
//实现函数
function sendBalance(bytes32 _id) {
  //任何调用该函数的行为都会触发SendBalance事件，并被JavaScript API检测到
  SendBalance(msg.sender, _id, msg.value);
}
}
```

使用JavaScript API来获取日志：

```
var abi = /* 编译后获取abi*/;
var EventExample = web3.eth.contract(abi);
var eventExample = EventExample.at(0x123 /* address */);
var event = eventExample.SendBalance ();
// 监听SendBalance事件是否被调用
event.watch(function(error, result){
  // result中会包含多种参数
  if (!error)
    console.log(result);
});
// 调用事件的另一种方式，调用后直接开始监听
var event = eventExample.SendBalance (function(error, result){
  if (!error)
    console.log(result);
});
});
```

6. 特殊变量

Solidity中有一些变量和函数是可以在全局范围内使用的，其中包括以太坊货币单位、时间单位、区块和交易属性、数学和加密函数等。

(1) 以太坊货币单位

数字后面可以加后缀，表示以太币货币的单位，如wei、finney或ether，相关单位之间可以转换。若以太币数量后没有加后缀，则默认单位是wei。货币单位也可以进行逻辑运算，如2 ether == 2000 finney，这个表达式计算结果为true。

```
pragma solidity ^0.4.0;
contract EtherExample{
  function isEqual() returns (bool){
    if (2 ether == 2000 finney){
      return true;
    }
    return false;
  }
}
```

(2) 时间单位

数字后面可以加秒（seconds）、分（minutes）、小时（hours）、天（days）、周（weeks）、年（years）作为后缀，并相互转换，默认以秒为单位。这些后缀不能用于声明变量，如果要传入变量，需要和常量转换。now则是当前时间戳。

```
pragma solidity ^0.4.0;
contract TimeExample{
    function f(uint start, uint daysAfter) {
        if (now >= start + daysAfter * 1 days) {
            //...
        }
    }
}
```

(3) 区块和交易属性

全局范围内有区块和交易属性，可以提供区块链当前的信息。

`block.blockhash(bytes32)`: 指定区块的哈希值;

`block.coinbase(address)`: 当前矿工的地址;

`block.difficulty(uint)`: 当前块的难度;

`block.gaslimit(uint)`: 当前块的gas值;

`block.number(uint)`: 当前区块的块号;

`block.timestamp(uint)`: 当前块的时间戳;

`msg.data(bytes)`: 完整的调用数据 (calldata);

`msg.gas(uint)`: 当前剩余gas值;

`msg.sender(address)`: 调用发起人的地址;

`msg.sig(bytes4)`: calldata的前4个字节 (即函数标识符);

`msg.value(uint)`: 所发送的消息中以太币的数量;

`now(uint)`: 当前块时间戳;

`tx.gasprice(uint)`: 交易的汽油价格;

`tx.origin(address)`: 交易发送方。

7. 合约部署

在实际项目开发中, 有很多种方式实现合约部署。如使用Console命令行的方式调用原生RPC接口部署智能合约, 或使用Console命令行的方式调用JavaScript API (web3.js) 部署智能合约, 或者使用Truffle框架自动化部署智能合约。

这里介绍一种使用JavaScript脚本实现合约的编译部署, 调用的接口同样是web3.js。使用该方式需要一点JavaScript编程基础。这里使用Multiply7合约来测试。合约代码如下:

```
contract Multiply7 {
    event Print(uint);
    function multiply(uint input) returns (uint) {
```



```

    Print(input * 7);
    return input * 7;
  }
}

```

把合约保存成一个以sol为后缀的文件，如Multiply7.sol，放到脚本文件所在的目录，同时开启以太坊私有链进行挖矿。编写部署脚本AutoDeploy.js如下：

```

var Web3 = require('web3');
var fs = require('fs');
//web3初始化
var web3;
if (typeof web3 !== 'undefined') {
  web3 = new Web3(web3.currentProvider);
}
else {
  web3 = new Web3(new
    Web3.providers.HttpProvider("http://localhost:8545"));
}
//从文件中读取Multiply7合约
//注意合约文件存放位置为当前目录
fs.readFile("./Multiply7.sol", function (error, result) {
  //打印读取的合约
  console.log(result.toString());
  //打印合约编译后的结果
  console.log("合约编译后结果:" +
    web3.eth.compile.solidity(result.toString()));
  //编译后获取abi
  var abiString =
    web3.eth.compile.solidity(result.toString()).Multiply7.info.abiDefinition;
  //编译后获取字节码code
  var code = web3.eth.compile.solidity(result.toString()).Multiply7.code;
  //new方法会执行两次，第一次是获得交易哈希，第二次是获得合约地址
  //使用abi和字节码可以部署一个合约到以太坊上
  web3.eth.contract(abiString).new({
    data: code,
    from: web3.eth.accounts[0],
    gas: 1000000
  }, function (error, myContract) {
    if (!myContract.address) {
      //获得交易哈希
      console.log(myContract.transactionHash);
    } else {
      //获得合约地址
      //myContract即为合约实例，用该实例可以调用合约方法
      console.log(myContract.address);
    }
  });
});

```

自动化部署脚本完成后，在终端中执行node AutoDeploy.js命令就能执行脚本文件，自动把Multiply7.sol合约部署到以太坊上。这种部署方式较为简单，相比Console控制台效率更高。

2.3.3 智能合约测试与执行

代码的编写完成并不意味着开发的结束，代码的测试也是非常重要的环节。智能合约作为一种特殊的计算机程序，也需要进行完善的测试。本节将会在Truffle中使用已经集成的Mocha测试框架对智能合约进行测试，测试环境可以是TestRPC或者geth。

2

1. Ethereum TestRPC

在第3章中将会使用以太坊客户端搭建私有链，所有的合约都可以使用该私有链来进行部署，然后调用合约方法完成一个DApp。但是在开发中发现，调用合约方法非常慢，这是因为以太坊私有链每执行一笔交易都需要挖矿确认，而挖矿是最消耗时间的了。另一种更快的创建测试网络的方法是使用TestRPC。TestRPC是基于Node.js开发的以太坊客户端，整个区块链的数据驻留在内存，可以模拟一个geth客户端的行为，包括所有的RPC API。发送给TestRPC的交易会被马上处理而不需要等待挖矿时间，及时响应，使测试开发工作更加方便。TestRPC可以在启动时帮你创建一堆存有资金的测试账户，因此更适合开发和测试。在合约开发初始阶段，可以使用TestRPC，随着合约不断完善，再部署到正式环境中去。

TestRPC安装使用命令行：

```
$ npm install -g ethereumjs-testrpc
```

安装完成后，在命令行中输入testrpc即可启动：

```
$ testrpc
```

TestRPC启动成功后的终端界面如下所示。

```
→ ~ testrpc
```

```
Scp256k1 bindings are not compiled. Pure JS implementation will be used.
```

```
EthereumJS TestRPC v3.0.2
```

```
Available Accounts
```

```
=====
```

```
(0) 0xd3803b205cad97a99cdee2e2b7019ed9b9370a78  
(1) 0xf3d1fb9dbcb4695274677dc7c0c7cf4ee71542ef  
(2) 0x83fba09f81c9dfe3993e61d32c06e2c7d04ad1  
(3) 0x1c0cef1579863444df1e9ca3539bace0b3a3d3dc  
(4) 0x7675204cdd6630f80a186b86f637be9fffb924a5b  
(5) 0x04e4567d54605e1091d41fb60b389115ec8432de  
(6) 0x8c9b00c5e7e66d699b393cf56fddd0299528822b  
(7) 0x9ec6045ed8af661a43062ec10e2f3fd25a7d91d7  
(8) 0xf0f3a34709067e0439109419d4cf427b93979c33  
(9) 0x525f78df9cf262660dac2d884f0beb6cac8024d8
```

```
Private Keys
```

```
=====
```

```
(0) ce8df7f7afe535009ab88aee71366e9df02680471245f7a0a44d94f05572c0066  
(1) 241f43ce8e8ed593a01406bccb647b18d114e973cad3ec0c700cdc82f7f73bde  
(2) 9efcaad3786a5ff062e12dc2dbc90d4e154c040bf4395c0ce8a49bd11feb8242  
(3) afcc0727c20b46ff8d0c1f97394b130d339dddca26168e169960e2a3055eac8
```

```

(4) 15b4a334533b8a76c0acb37dd2937f0ffa458f9370b617a565756e6a50ab86d7
(5) 483a3248c4539eb69aa52f5e5a54ea9e68651c2e4010f83d0efd2e06661321ff
(6) 19a3ab3cd8127db5b8cd5e070293915e0e7ca6227278f351c897673b8b3a7edd
(7) dc017348df68557f7eae267d0f8b4e23966bf46233a3feef31e87e49da9d7de5
(8) ae125c9f0e7e973b2626102b9fc742304d3d7f52e0fc10120dd3929222c25a7f
(9) c8bddcd6d5f266d9be53f3bf793c552e9a0b1e15fa19e5348f32decf85680ba3
HD Wallet
=====
Mnemonic:      risk chimney oven great pony ancient donor pass tide
win voice aware
Base HD Path:  m/44'/60'/0'/0/{account_index}
Listening on localhost:8545

```

可以看到，TestRPC默认会创建10个账户，并显示账户的公私钥，默认端口号同geth创建的私有链相同，都为8545。同时TestRPC也可以在启动时指定参数选项，此时的启动命令如下。

```
$ testrpc <options>
```

Options参数如下。

- (1) -a或--accounts: 指定在启动时需要生成的账户数量；
- (2) -b或--blocktime: 指定自动挖矿时间，默认是0，不进行自动挖矿；
- (3) -d或--deterministic: 基于预定义助记符生成确定账号；
- (4) -n或--secure: 默认锁定账号（对于第三方交易签名较为有用）；
- (5) -m或--mnemonic: 使用指定的钱包助记符来生成初始化地址；
- (6) -p或--port: 监听端口号，默认为8545；
- (7) -h或--hostname: 监听的主机名，默认为Node的server.listen()；
- (8) -s或--seed: 使用随机数据来生成钱包助记符；
- (9) -g或--gasPrice: 使用的gas价格，默认为200000000000；
- (10) -l或--gasLimit: 使用的gas限制，默认为0x47E7C4；
- (11) -f或--fork: 从一个指定区块号已经运行的以太坊客户端fork，输入应当是HTTP地址，端口号是其他的客户端，如http://localhost:8545，也可以使用@符号指定需要fork的区块号，如http://localhost:8545@12684；

- (12) --debug: 输出虚拟机调试操作码。

目前TestRPC实现的RPC方法实现如下：

```

eth_accounts
eth_blockNumber
eth_call
eth_coinbase
eth_compileSolidity

```

```

eth_estimateGas
eth_gasPrice
eth_getBalance
eth_getBlockByNumber
eth_getBlockByHash
eth_getCode (only supports block number "latest")
eth_getCompilers
eth_getFilterChanges
eth_getFilterLogs
eth_getLogs
eth_getStorageAt
//...

```

2. Truffle测试实例分析

目前区块链数据还很难像数据库一样实现全部数据的可视化,并且不能直接操作修改区块链中的数据,所以对合约的测试存在一定的困难。目前测试合约的主要方法还是通过调用合约函数,根据输入输出结果来进行测试。常用的测试方法有:使用在线编译器Remix来可视化的调用方法,非常方便高效;或者通过交互命令行的方式,分别使用JSON RPC和JavaScript API接口来执行合约方法,根据输出数据的结果判断调用是否成功。以上方式都实现了对智能合约的测试与执行。

在Truffle框架中,已经集成了自动化测试框架Mocha,并且断言使用的是Chai。使用这两个库可以非常方便地编写合约的自动化测试代码。按照正常规范的开发流程,这些测试代码应该在编写合约的同时完成。

创建完一个Truffle项目后,在目录下有一个test文件夹,里面存放的就是对合约的测试代码,项目创建完成已经默认有一个测试文件。在该目录下可以建立多个测试文件,文件后缀需要为.js、.es、.es6、.jsx,其他后缀的文件在测试过程中会被忽略。下面将介绍如何使用Truffle测试框架实现对合约的测试。使用的例子是Truffle默认生成的合约MetaCoin。测试文件命名为metacoin.js,测试脚本实现如下。

```

//每次执行contract()函数,合约都会在以太坊客户端重新部署。所以上一次测试的结果不会遗留到下一次
contract('MetaCoin', function (accounts) {
  it("初始化10000MetaCoin给以太坊第一个账户", function () {
    //meta即为当前部署合约的实例
    var meta = MetaCoin.deployed();

    return meta.getBalance.call(accounts[0]).then(function (balance) {
      //断言失败则会打印message
      assert.equal(balance.valueOf(), 10000, "第一个账户初始化10000失败");
    });
  });
  it("发送MetaCoin", function () {
    var meta = MetaCoin.deployed();
    //分别以当前以太坊客户端中第一个、第二个账户来做测试
    var account_one = accounts[0];
    var account_two = accounts[1];

    //发送账户的初始余额

```



```

var account_one_starting_balance;
//接收账户的初始余额
var account_two_starting_balance;
//交易结束后发送账户的余额
var account_one_ending_balance;
//交易结束后接收账户的余额
var account_two_ending_balance;
var amount = 1000;
//分别获取两个账户的余额
return meta.getBalance.call(account_one).then(function (balance) {
    account_one_starting_balance = balance.toNumber();
    return meta.getBalance.call(account_two);
}).then(function (balance) {
    account_two_starting_balance = balance.toNumber();
    //调用发送MetaCoin合约方法
    return meta.sendCoin(account_two, amount, { from:account_one});
}).then(function () {
    return meta.getBalance.call(account_one);
}).then(function (balance) {
    account_one_ending_balance = balance.toNumber();
    return meta.getBalance.call(account_two);
}).then(function (balance) {
    account_two_ending_balance = balance.toNumber();

    //以下两个断言只要有一个失败，则表示本次发送MetaCoin交易失败
    assert.equal(account_one_ending_balance, account_one_starting_balance - amount,
        "发送者发送MetaCoin错误");
    assert.equal(account_two_ending_balance, account_two_starting_balance + amount,
        "接收者接收MetaCoin错误");
    });
});
});

```

执行测试脚本同样需要开启以太坊私有链，推荐使用TestRPC，在终端下可以使用以下命令来执行自动化部署脚本。

```

//执行test文件夹下所有测试脚本
$ truffle test
//执行某一个测试脚本
$ truffle test metacoin.js

```

上述代码中有两个it()代码块，分别表示两个测试用例。执行过程首先是编译部署合约，然后执行每一个it()代码块，同时会显示每个代码块执行时间和整个测试过程的执行时间，最终提示测试通过，显示如下：

```

→ myTruffle git:(master) X truffle test
Contract: MetaCoin
✓ 初始化10000MetaCoin给以太坊第一个账户 (38ms)
✓ 发送MetaCoin (120ms)
2 passing (339ms)

```

接下来，我们改变第二个测试用例中发送的MetaCoin值，将其设置为1000000（超过账户所

拥有的MetaCoin额度), 进行第二次测试。发现第二个测试用例没有通过, 会得到AssertionError信息提示, 显示如下:

```
→ myTruffle git:(master) X truffle test
Contract: MetaCoin
  ✓ 初始化10000MetaCoin给以太坊第一个账户
  1) 发送MetaCoin
Events emitted during test:
-----
Transfer(from: 0xd3803b205cad97a99cdee2e2b7019ed9b9370a78, message: MetaCoin不足,
  发送失败)
-----
  1 passing (295ms)
  1 failing
  1) Contract: MetaCoin发送MetaCoin:
AssertionError: 发送者发送MetaCoin错误: expected 10000 to equal -990000
at test/metacoin.js:43:20
at process._tickDomainCallback
(internal/process/next_tick.js:129:7)
```

由于要发送1000000 MetaCoin, 在断言中第一个账户的结果余额不等于初始余额减去发送的MetaCoin值, 所以断言失败。从错误提示中看到, 接收到合约发出的event事件, 可以方便地查看到更多的调试信息。所以在一些重要的方法中使用event事件将有助于后续的调试。

Truffle官方建议自动化测试在EthereumJS TestRPC客户端下进行, 因为这样比使用以太坊私链更快。而且, Truffle还可以充分利用TestRPC中的一些特性在运行时提升速度。所以, 作为一个更加通用的工作流, Truffle建议在开发和测试过程中使用TestRPC, 然后再运行测试到geth或其他官方以太坊客户端。

2.3.4 智能合约实例分析

下面我们挑选一个合约实例来进行详细的分析, 这个合约较为复杂, 但展示了很多Solidity的特性。这个例子是关于银行积分的合约, 银行可以发行积分给客户, 客户之间可以转让积分, 客户可以使用积分到积分商城兑换商品, 并能提供基本的查询功能。关于积分合约的详细实现以及与接口的交互, 可以参考本书8.1节基于以太坊的通用积分系统案例分析。

首先把以下合约部署到浏览器编译器或以太坊私有链上, 然后进行调用测试。

```
contract Score{
  address owner; //合约的拥有者即银行
  uint issuedScoreAmount; //银行已经发行的积分总数
  uint settledScoreAmount; //银行已经清算的积分总数
  struct Customer {
    address customerAddr; //客户address
    bytes32 password; //客户密码
    uint scoreAmount; //积分余额
    bytes32[] buyGoods; //购买的商品数组
  }
}
```

```

struct Good {
    bytes32 goodId; //商品Id;
    uint price; //价格;
    address belong; //商品属于哪个商户address;
}
mapping (address=>Customer) customer; //根据客户address查找
mapping (bytes32=>Good) good; //根据商品Id查找该件商品
address[] customers; //已注册的客户数组
bytes32[] goods; //已经上线的商品数组
//增加权限控制, 某些方法只能由合约的创建者调用
modifier onlyOwner(){
    if(msg.sender != owner) throw;
    _;
}
//构造函数
function Score() {
    owner = msg.sender;
}
//返回合约调用者地址
function getOwner() constant returns(address) {
    return owner;
}
//string类型转化为bytes32类型
function stringToBytes32(string memory source)constant internal returns (bytes32 result) {
    assembly {
        result := mload(add(source, 32))
    }
}
//注册一个客户
event NewCustomer(address sender, bool isSuccess, string message);
function newCustomer(address _customerAddr,
    string _password) {
    //判断是否已经注册
    if(!isCustomerAlreadyRegister(_customerAddr)) {
        //还未注册
        customer[_customerAddr].customerAddr = _customerAddr;
        customer[_customerAddr].password = stringToBytes32(_password);
        customers.push(_customerAddr);
        NewCustomer(msg.sender, true, "注册成功");
        return;
    }else {
        NewCustomer(msg.sender, false, "该账户已经注册");
        return;
    }
}
//判断一个客户是否已经注册
function isCustomerAlreadyRegister(address _customerAddr)internal
returns(bool) {
    for(uint i = 0; i < customers.length; i++) {
        if(customers[i] == _customerAddr) {
            return true;
        }
    }
    return false;
}

```

```

}
//银行发送积分给客户，只能被银行调用，且只能发送给客户
event SendScoreToCustomer(address sender, string message);
function sendScoreToCustomer(address _receiver,
    uint _amount)onlyOwner {
    if(isCustomerAlreadyRegister(_receiver)) {
        //已经注册
        issuedScoreAmount += _amount;
        customer[_receiver].scoreAmount += _amount;
        SendScoreToCustomer(msg.sender, "发行积分成功");
        return;
    }else {
        //还没注册
        SendScoreToCustomer(msg.sender, "该账户未注册，发行积分失败");
        return;
    }
}
//根据客户 address 查找余额
function getScoreWithCustomerAddr(address customerAddr)constant
returns(uint) {
    return customer[customerAddr].scoreAmount;
}
//两个账户转移积分，任意两个账户之间都可以转移
event TransferScoreToAnother(address sender, string message);
function transferScoreToAnother(uint _senderType,
    address _sender,
    address _receiver,
    uint _amount) {
    string memory message;
    if(!isCustomerAlreadyRegister(_receiver)) {
        //目的账户不存在
        TransferScoreToAnother(msg.sender, "目的账户不存在，请确认后再转移!");
        return;
    }
    if(_senderType == 0) {
        //客户转移
        if(customer[_sender].scoreAmount >= _amount) {
            customer[_sender].scoreAmount -= _amount;
            if(isCustomerAlreadyRegister(_receiver)) {
                customer[_receiver].scoreAmount += _amount;
            }
            TransferScoreToAnother(msg.sender, "积分转让成功!");
            return;
        }else {
            TransferScoreToAnother(msg.sender, "你的积分余额不足，转让失败!");
            return;
        }
    }
}
//银行查找已经发行的积分总数
function getIssuedScoreAmount()constant returns(uint) {
    return issuedScoreAmount;
}

```



```

//银行查找已经清算的积分总数
function getSettledScoreAmount()constant returns(uint) {
    return settledScoreAmount;
}
//用户用积分购买一件商品
event BuyGood(address sender, bool isSuccess, string message);
function buyGood(address _customerAddr, string _goodId) {
    //首先判断输入的商品Id是否存在
    bytes32 tempId = stringToBytes32(_goodId);
    //该件商品已经添加, 可以购买
    if(customer[_customerAddr].scoreAmount < good[tempId].price) {
        BuyGood(msg.sender, false, "余额不足, 购买商品失败");
        return;
    }else {
        customer[_customerAddr].scoreAmount -= good[tempId].price;
        //对应的商户增加相应的余额
        customer[_customerAddr].buyGoods.push(tempId);
        BuyGood(msg.sender, true, "购买商品成功");
        return;
    }
}
}

```

2.4 以太坊重大事件与现存问题

随着以太坊的逐渐流行, 各国政府和央行开始研究发行自己的“数字货币”, 托管在以太坊上的项目越来越多, 这期间, 以太坊发生过几个重大事件。本节将对这些重要事件进行剖析, 从而加深读者对以太坊的理解。

2.4.1 The DAO 攻击事件

The DAO是以太坊最大的众筹项目, 作为去中心化自治组织, 其目的是为组织规则以及决策机构编写代码, 从而消除书面文件的需要, 减少管理人员, 从而创建出一个去中心化的管理架构。The DAO项目于2016年4月30日开始, 众筹窗口开放了28天。截止到项目结束, 筹得1.5亿美元, 共超过11 000位成员参与, 成为历史上最大的众筹项目。

The DAO创始人之一Stephan Tual于6月12日宣布, 发现了软件中存在的递归调用漏洞问题, 但这对DAO资金来说不会有影响, 所以这个问题很快被压了下来, 因为DAO正处于测试阶段。在程序员修复这一漏洞及其他问题期间, 一个不知名黑客开始利用这一途径收集The DAO代币销售中所得的以太币。6月18日, 黑客利用The DAO智能合约中一个splitDAO函数的漏洞, 不断从The DAO项目资产池中分离DAO资产给自己。黑客成功挖到超过360万个以太币。当时以太币价格从20多美元直接跌破13美元。很多人都在尝试从The DAO脱离出来, 以防止以太币被盗, 但是他们无法在短期内获得所需票数。The DAO持有近15%的以太币总数, 因此这次攻击对以太坊产生了重大的负面影响。对于The DAO受到攻击的解决方法, 有两种建议。

1. 软分叉提议

以太坊基金会的Vitalik Buterin在以太坊官方博客发布题为“紧急状态更新:关于DAO的漏洞”的文章,解释了被攻击的细节以及解决方案提议。提议方案为进行一次软分叉,不会有回滚,不会有任何交易或者区块被撤销。软分叉将从块高度1 760 000开始,把任何与The DAO和child DAO相关的交易认作无效交易,以此阻止攻击者在27天之后提走被盗的以太币。之后会有一次硬分叉将以以太币找回。上述文章公布后,攻击暂时停止。但是随后在6月19日,自称攻击者的人通过网络匿名访谈宣布,会通过智能合约的形式奖励不支持软分叉的矿工100万以太币和100比特币,来对抗以太坊基金会提议的软分叉。随后攻击再次开始,但是只有少量DAO被分离。6月22日,白帽黑客开展罗宾汉行动,将The DAO资产转移到安全的子DAO中。随后黑帽黑客(攻击者)开始攻击白帽黑客所创建的为安全转移The DAO资产的智能合约。

2. 硬分叉提议

该提议是Stephan Tual提出的,要求矿工彻底解除盗窃并且归还The DAO所有以太币,这样就能自动归还给代币持有人,从而结束The DAO项目。Stephan Tual对软分叉是反对的,他指出,如果27天都用软分叉,那么攻击者就不能索回他放到子DAO中的资金了。应该使用硬分叉追回所有以太币,包括DAO的“额外余额”以及被盗资金,归还到智能合约上。这个智能合约将包含一个简单函数: `withdraw()`。这样每个人都可能参与到DAO,以提取他们的资金。

2.4.2 以太坊现存问题

以太坊作为目前在区块链上较为成功的开源项目,还处于快速发展和探索之中。随着在以太坊上构建的项目越来越多,很多问题也逐渐暴露出来,比如共识效率过低的问题,隐私可能遭到泄露的问题,以及大规模的数据存储问题。这些问题将是之后构建大型DApp不得不面对的难题。

1. 共识效率低下

目前以太坊正处于第二个阶段Homestead,以太坊的前三个阶段Frontier、Homestead、Metropolis都采用PoW工作量证明的共识算法,在第四阶段Serenity将会使用PoS的共识算法。

PoW是一种非常有效的共识机制,比特币网络使用的也是PoW的共识机制。但是在构建大型商业去中心化应用中,这种共识机制在效率方面的弊端非常明显。PoW共识机制获得货币量的多少取决于挖矿工作的成效,用户所使用的计算机性能越好,挖矿获得的货币就越多,根据工作量分配货币。这样存在的问题:一是目前比特币已经吸引了全球大部分的算力,使用PoW共识机制的区块链应用很难获得相同的算力来保障自身的安全,可能存在一定的风险;二是挖矿造成大量的资源浪费,尤其是电力资源的浪费,使维护这种共识机制的成本过高;三是共识达成的周期较长,平均要14秒打包一个区块,这个时间对于商业化项目来说延迟过长,不能真正符合商业应用的要求。

同时, PoW还受到一个小概率事件的影响,那就是区块链分叉。当网络上有2个或以上节点

同时竞争到记账权力时，在网络中就会产生2个或以上的区块链分支，此时到底哪个分支记录的数据是有效的，则要再等下一个记账周期，最终由最长的区块链分支来决定，因此交易数据有较大延迟。这种情况也可能导致在较短链上的数据丢失。

2. 隐私保护缺乏

对于以太坊公有链来说，虽然有一定的匿名性，但是区块上的交易账本是完全公开的，每个节点上都有一份完整的账本。并且由于区块链计算余额、验证交易有效性等都需要追溯每一笔账，因此交易数据都是公开透明的。如果知道某个人的账户，就能知道他的交易记录和余额，没有隐私可言。所以如果金融类应用直接使用以太坊公有链，显然是缺乏隐私性的。同时在联盟链中，多个企业应该相互信任并共享数据，但是万一发生数据泄露，则很难追踪是从哪一个企业泄露的。

目前对于隐私保护已经有多种解决方式：混币、同态加密、零知识证明等。

(1) 混币。在多人参与的大量交易中，分隔输入输出地址之间的关系，使其不能找到一一对应的交易，相当于做一个混淆。多次使用混币，能有效提高隐私保护的力度。

(2) 同态加密。常用在公有链和联盟链中，是一种无需对加密数据进行提前解密就可以执行计算的方法。能够让公有链起到类似私有链的隐私效果，其本身不会对区块链进行任何重大的修改。

(3) 零知识证明。零知识证明是可以在无需泄露数据本身的情况下证明某些数据运算的一种密码学技术。在双方的交易记录中，证明某些数据是真实的，而无需泄露其他额外的信息，减少了较多数据暴露的风险。

3. 大规模存储困难

无论是全球比特币网络还是以太坊，随着交易量的增大，大规模的数据存储问题也慢慢出现。由于每个节点都有一份完整账本，区块链系统的数据存储存在非常多的冗余，并且有时要追溯每一笔交易，因此，随着时间推进，当交易数据超大的时候，就会有性能问题。如第一次使用需要下载历史上所有交易记录才能正常工作，那么首次交易的执行时间会较长。每次交易时，为了验证你确实拥有足够的钱而需要追溯每一笔历史交易来计算余额，当整条链过长的时候，这个问题是明显存在的。

同时，由于目前去中心化应用的运行依靠的是智能合约，当一个应用中所包含的数据过多时，智能合约的升级与应用的数据迁移也会遇到问题。如何保证大规模数据迁移的可靠性也是业内研究的重点。

4. 信息难以监管

与比特币平台类似，以太坊作为一种公有链，任何用户都以一串无意义的数字作为唯一标识而参与交易，以太坊区块中记录的交易信息是匿名化的，资金的流向非常难以监管，导致很多非法交易都通过比特币、以太坊等“数字货币”进行支付，这些流通的“数字货币”可能成为洗钱和非法融资的工具。这一问题大大限制了以太坊区块链平台商业化应用的发展，特别是涉及政府

部门、金融机构、大型企业等的核心业务时，目前的以太坊平台基本无法满足需求。只有加入了权限控制的企业级联盟区块链平台（例如Hyperchain），才能较好地解决信息难以监管的难题。

2.5 本章小结

本章对以太坊的发展历史、基本概念、客户端、账户管理及以太坊网络等的基础知识进行了介绍，并对以太坊共识机制、虚拟机、数据存储和加密算法等以太坊关键模块的核心原理进行了剖析，详细介绍了以太坊智能合约的编写、部署、测试与执行，最后对以太坊发展过程中的重大事件和目前存在的主要问题进行了分析探讨。

第3章

以太坊应用开发基础

3

本章将介绍如何从零开始开发一个以太坊DApp（去中心化应用）。首先，讲解如何配置以太坊环境、搭建以太坊私有链，私有链可作为DApp的运行环境。然后，介绍集成开发环境Mix和浏览器实时编译器（Real-time Compiler），DApp的大部分开发工作（如智能合约的编写）都可以在这些IDE中完成。之后，给出以太坊智能合约与前端交互的两种重要接口：JSON RPC和JavaScript API，通过这些接口可以调用以太坊中的智能合约。接着，介绍当前比较成熟的快速开发DApp的框架（Meteor和Truffle），并且给出一套分层可扩展的项目开发流程。最后，实现一个以太坊应用MetaCoin，方便读者完整地学习使用框架开发DApp，完成智能合约的编写、部署、调用以及与前端页面的测试交互。

3.1 以太坊开发环境搭建

本节讲述的以太坊环境配置和私有链搭建都是在Mac下进行的，不同操作系统的配置大同小异，推荐在Mac或Ubuntu下搭建以太坊私有链。

3.1.1 配置以太坊环境

• 安装Go环境

因为以太坊是用Go语言进行开发的，所以要在本机上安装以太坊，首先需要安装Go的环境。进入<https://golang.org/dl/>下载对应的Go语言安装包。如果是Mac，则下载go1.7.4.darwin-arm64.pkg，双击安装即可。默认安装在/usr/local/go目录下，并自动设置了环境变量。

同时还需要配置一个GOPATH环境变量，作为Go的工作目录。进入终端编辑.bash_profile文件：

```
vi ~/.bash_profile
```

加入以下环境变量：

```
#go
export GOPATH=/usr/local/go
export GOBIN=$GOPATH/bin
export PATH=$PATH:$GOBIN
```

若要配置文件立即生效，在终端执行以下命令：

```
source ~/.bash_profile
```

在终端执行以下命令，可查看是否安装成功：

```
go version
```

若出现如下的命令行，则表示Go语言编程环境安装成功：

```
→ go version
go version go1.7.3 darwin/amd64
```

• 安装Node.js、npm

npm是Node.js下的一个包管理工具，可以非常方便地安装一些基于JavaScript的软件和包。基于以太坊的很多开发工具也都是基于JavaScript来开发的，可以使用npm进行安装。进入<https://nodejs.org/en/>，网站会根据操作系统提示下载不同的Node.js版本，下载后安装即可。默认会同时安装Node.js和npm。终端执行以下命令，可查看是否安装成功：

```
go version
```

若出现如下的命令行则表示安装成功：

```
→ npm -v
3.10.9
→ node -v
v6.9.1
```

• 安装以太坊Ethereum

进入终端，执行以下命令：

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install ethereum
```

执行以下命令可查看以太坊是否安装成功：

```
geth version
```

若出现如下的命令行则表示安装成功：

```
→ geth version
Geth
Version: 1.5.2-stable
Git Commit: c8695209f609375ceb06e8f4151dc9093f38cac5
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.7.3
OS: darwin
GOPATH=
GOROOT=/usr/local/Cellar/go/1.7.3/libexec
```

- 安装solc编译器

solc是智能合约Solidity的编译器，在终端执行以下命令安装solc：

```
npm install solc
```

执行以下命令可查看solc是否安装成功：

```
solc --version
```

若出现如下的命令行则表示solc安装成功：

```
→ solc --version
solc, the solidity compiler commandline interface
Version: 0.4.4+commit.4633f3de.Darwin.appleclang
```

3.1.2 搭建以太坊私有链

- 创建账户（公钥）

在终端输入以下命令3次，可以创建3个以太坊账户，在创建时需要输入该账户的密码：

```
geth account new
```

成功创建一个账户的命令行如下所示：

```
→ geth account new
Your new account is locked with a password. Please give a password. Do not forget this password.
Passphrase:
Repeat passphrase:
Address: {b1ad0442705ff62e3f3fa53cdea69a4d500de3fd}
```

所有的账户都可以在Ethereum安装目录的keystore下查看到。

- 编写创世块文件

在根目录（~/）下创建test-genesis.json文件。可以通过设置alloc中账户地址的balance，给刚刚申请的账户分配足够多的以太币。

```
{
  "nonce": "0x00000000000000042",
  "difficulty": "0x1",
  "alloc": {
    "b6354aaa21390856f039ac23c73baea558796e05": {
      "balance": "2000000980000000000000000000000"
    },
    "81063419f13cab5ac090cd8329d8fff9feed4a0": {
      "balance": "2000000980000000000000000000000"
    },
    "9da26fc2e1d6ad9fdd46138906b0104ae68a65d8": {
      "balance": "2000000980000000000000000000000"
    },
    "bd2d69e3e68e1ab3944a865b3e566ca5c48740da": {
      "balance": "2000000980000000000000000000000"
    }
  }
}
```

```

    },
    "ca9f427df31a1f5862968fad1fe98c0a9ee068c4": {
      "balance": "2000000980000000000000000000000000"
    }
  },
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "timestamp": "0x00",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbbdb7a38e1e50b1b82fa",
  "gasLimit": "0xb2d05e00"
}

```

● 初始化创世块

Ethereum默认安装在“~/Library/Ethereum”目录下，使用以下命令来初始化刚刚创建的创世块文件。

```
geth --datadir "~/Library/Ethereum" init ~/test-genesis.json
```

初始化成功的命令行结果如下所示：

```

→ geth --datadir "~/Library/Ethereum" init ~/test-genesis.json
I0608 23:46:04.811565 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to
/Users/chenyufeng/Library/Ethereum/geth/chaindata
I0608 23:46:04.823181 ethdb/database.go:176] closed
db:/Users/chenyufeng/Library/Ethereum/geth/chaindata
I0608 23:46:04.823250 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to
/Users/chenyufeng/Library/Ethereum/geth/chaindata
I0608 23:46:04.868505 core/genesis.go:92] Genesis block already in chain. Writing canonical number
I0608 23:46:04.868600 cmd/geth/main.go:256] successfully wrote genesis block
and/or chain rule set:
eec1acd42c7f56b3919c3e853a605c611230414cca1bfc564ee91de4fdaffaab

```

● 配置自动解锁账户的脚本

进入Ethereum安装目录“~/Library/Ethereum”，创建password文件，并在该文件中输入在test-genesis.json中每个账户对应的密码，每个密码一行，只需要输入密码即可。如下所示：

```

123456
123456
123456

```

● 编写以太坊启动脚本

创建启动脚本文件private_blockchain.sh文件，并在文件中配置如下内容：

```

geth --rpc --rpcaddr="0.0.0.0" --rpccorsdomain="*" --unlock '0,1,2' --password
~/Library/Ethereum/password --nodiscover --maxpeers
'5' --networkid '1234574' --datadir '~/Library/Ethereum' console

```

以后每次启动geth节点时，只需要在终端执行以下命令即可：

```
sh private_blockchain.sh
```


成功启动以太坊私有链的结果如下述命令行所示：

```
➔ sh private_blockchain.sh
I0608 23:47:43.436154 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to
/Users/chenyufeng/Library/Ethereum/geth/chaindata
I0608 23:47:43.446379 ethdb/database.go:176] closed
db:/Users/chenyufeng/Library/Ethereum/geth/chaindata
I0608 23:47:43.447077 node/node.go:175] instance:
Geth/v1.5.2-stable-c8695209/darwin/go1.7.3
I0608 23:47:43.447439 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to
/Users/chenyufeng/Library/Ethereum/geth/chaindata
I0608 23:47:43.493469 eth/backend.go:193] Protocol Versions: [63 62], Network Id: 1234574
I0608 23:47:43.496303 core/blockchain.go:214] Last header: #10924 [5f198d91...] TD=18451334389
I0608 23:47:43.496324 core/blockchain.go:215] Last block: #10924 [5f198d91...] TD=18451334389
I0608 23:47:43.496330 core/blockchain.go:216] Fast block: #10924 [5f198d91...] TD=18451334389
I0608 23:47:43.499846 p2p/server.go:336] Starting Server
I0608 23:47:43.500257 p2p/server.go:604] Listening on [::]:30303
I0608 23:47:43.502038 node/node.go:340] IPC endpoint opened:
/Users/chenyufeng/Library/Ethereum/geth.ipc
I0608 23:47:43.502395 node/node.go:410] HTTP endpoint opened:
http://0.0.0.0:8545
I0608 23:47:44.404571 cmd/geth/accountcmd.go:200] Unlocked account
90c2323cdeff75fd82e65ac496fc45eafadf4563
Welcome to the Geth JavaScript console!

instance: Geth/v1.5.2-stable-c8695209/darwin/go1.7.3
coinbase: 0x90c2323cdeff75fd82e65ac496fc45eafadf4563
at block: 10924 (Fri, 19 May 2017 23:48:15 CST)
  datadir: /Users/chenyufeng/Library/Ethereum
  modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0
  txpool:1.0 web3:1.0
```

● 启动挖矿

以太坊上执行每一笔交易都需要矿工挖矿后才能被确认。在私有链上同样可以执行挖矿操作，在启动私有链后执行以下命令开始挖矿。

```
miner.start()
```

执行以下命令停止挖矿：

```
miner.stop()
```

挖矿示意如下述命令行所示：

```
> miner.start()
I0608 23:49:41.789728 eth/backend.go:479] Automatic pregeneration of ethash DAG ON (ethash dir:
/Users/chenyufeng/.ethash)
I0608 23:49:41.789732 miner/miner.go:136] Starting mining operation (CPU=4 TOT=5)
true
> I0608 23:49:41.789794 eth/backend.go:486] checking DAG (ethash dir:
/Users/chenyufeng/.ethash)
I0608 23:49:41.790108 miner/worker.go:542] commit new work on block 10925 with 0 txs & 0 uncles.
Took 336.858µs
I0608 23:49:41.790164
```

```

vendor/github.com/ethereum/ethash/ethash.go:259] Generating DAG for epoch 0 (size 1073739904)
(0000000000000000000000000000000000000000000000000000000000000000)
I0608 23:49:42.630374
vendor/github.com/ethereum/ethash/ethash.go:276] Done generating DAG for epoch 0, it took
840.206482ms
I0608 23:50:34.233461 miner/worker.go:344] Mined block (#10925 / 139b574d).
Wait 5 blocks for confirmation
I0608 23:50:34.233701 miner/worker.go:542] commit new work on block 10926 with 0 txs & 0 uncles.
Took 198.171µs
I0608 23:50:34.368638 miner/worker.go:542] commit new work on block 10926 with 0 txs & 0 uncles.
Took 1.283619ms
I0608 23:50:45.947720 miner/worker.go:344] Mined block (#10926 / 03662410).
Wait 5 blocks for confirmation
I0608 23:50:45.947933 miner/worker.go:542] commit new work on block 10927 with 0 txs & 0 uncles.
Took 171.658µs
I0608 23:50:46.024143 miner/worker.go:542] commit new work on block 10927 with 0 txs & 0 uncles.
Took 204.575µs

```

3.2 以太坊集成开发环境

一般软件项目的开发都依赖于集成开发环境 (IDE), 以太坊去中心化应用的开发也有专用的开发环境: Mix和浏览器实时编译器。目前在实际的项目开发中, 使用最为广泛的还是浏览器实时编译器, 因为其编译快速, 错误提示友好, 而且测试方便。

3.2.1 Mix IDE

Mix是以太坊官方的DApp集成开发环境, 用以在以太坊平台上快速开发应用。可以用于合约的编写、测试和部署到区块链, 同时进行用户界面的开发。webthree-umbrella是以太坊官方的项目, 开发语言为C++, 里面主要包括3个客户端: AlethZero、Mix和Eth, 可以运行在Windows、Linux和macOS操作系统上。webthree-umbrella其实是cpp-ethereum项目的一个shell, 是以太坊基金会的前CTO Gavin Wood主持编写的。

进入<https://github.com/ethereum/webthree-umbrella/releases>可以下载不同操作系统的Mix版本, 安装使用方法非常简单。下载目录如图3.1所示。

Downloads

📦 cpp-ethereum-osx-elcapitan-v1.2.9.dmg	78 MB
📦 cpp-ethereum-osx-elcapitan-v1.2.9.zip	11.6 MB
📦 cpp-ethereum-osx-yosemite-v1.2.9.dmg	81 MB
📦 cpp-ethereum-osx-yosemite-v1.2.9.zip	12.6 MB
📦 cpp-ethereum-windows-v1.2.9.zip	32.1 MB
📄 Source code (zip)	
📄 Source code (tar.gz)	

图3.1 下载Mix

● 创建应用

打开Mix，整体操作界面如图3.2所示。

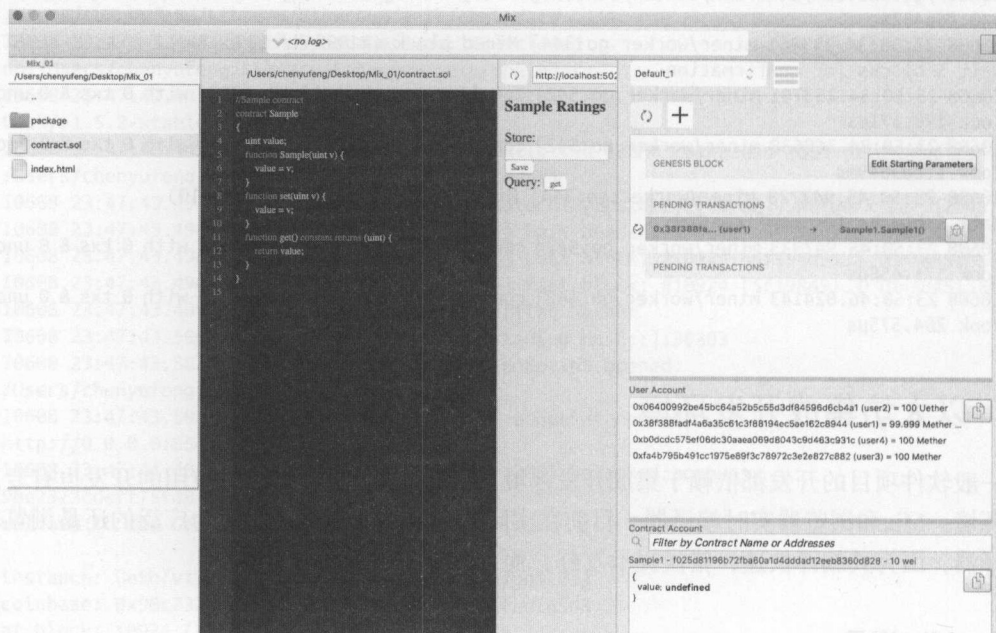


图3.2 Mix主界面

Mix已经默认创建了一个非常简单的智能合约Sample:

```
//Sample合约
contract Sample
{
    uint value;
    function Sample(uint v) {
        value = v;
    }
    function set(uint v) {
        value = v;
    }
    function get() constant returns (uint) {
        return value;
    }
}
```

Sample合约使用set(uint v)方法为合约中的状态变量value设置值，使用get()方法取出value的值。构造方法Sample(uint v)会在合约初始化的时候被调用。

● 部署应用

点击菜单栏Deploy→Deploy to Network，配置部署相关信息。首先需要在终端中启动以太坊

私链，并开启挖矿。如图3.3所示，在Deploy Scenario中，Deployment account是调用部署合约方法的账户，当前以太坊私链中的账户都可以进行选择；在Gas price中可以选择默认的gas值；Deployment cost是预计要花费的以太币的值；Deployed Contracts是部署合约完成后，合约在以太坊中的合约地址；Verifications是确认，合约在部署过程中需要被区块确认，经过多个区块挖矿后，合约才能部署成功。

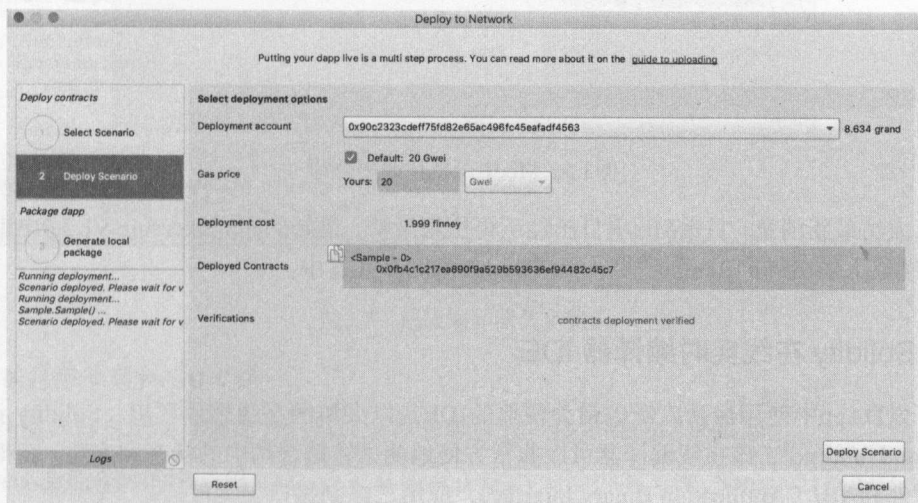


图3.3 Mix中部署应用

● 调用合约

可以通过主界面右侧视图编辑调用合约的参数，也可以在合约创建后在构造函数中赋值value，如图3.4所示。

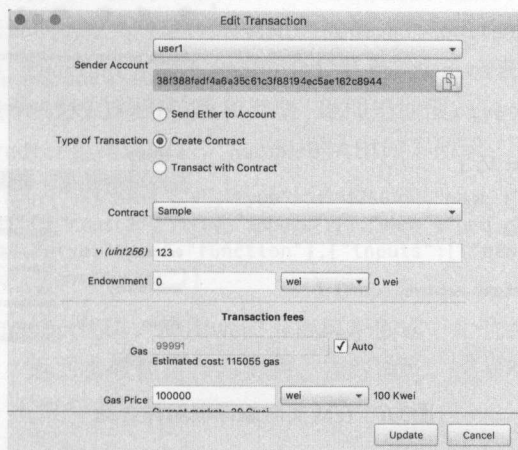


图3.4 Mix中调用合约

也可以在应用提供的前端界面中操作合约方法, 设置set或获得get合约中的状态变量value, 如图3.5所示。

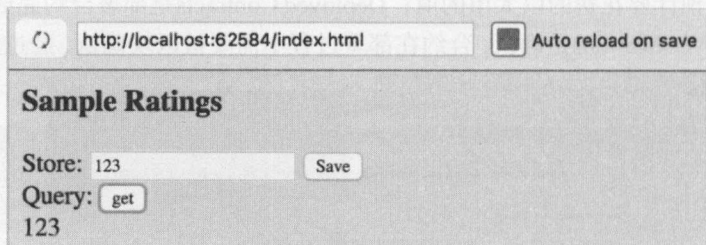


图3.5 Mix中可视化调用合约

据以太坊最新消息, 目前Mix项目已经不再更新维护, 最新的cpp-ethereum-v1.3.0中也没有了Mix。后续以太坊会转向Remix项目, 目前Remix仍旧在开发中。

3.2.2 Solidity 在线实时编译器 IDE

在开发DApp中使用最为广泛也最为便捷的IDE是以太坊的在线编译环境: Solidity real-time compiler and runtime^①。该在线编译器可以非常方便地测试智能合约中的状态和方法, 获得合约的字节码文件和ABI (Application Binary Interface, 应用二进制接口) 文件。

● 编译智能合约

这里使用3.2.1节中的Sample合约在在线编译器中进行编译测试。主界面如图3.6所示, 在线编译器的左侧是代码编辑区域, 可以直接在里面进行智能合约的编写, 右侧是进行参数输入输出和调试区域。

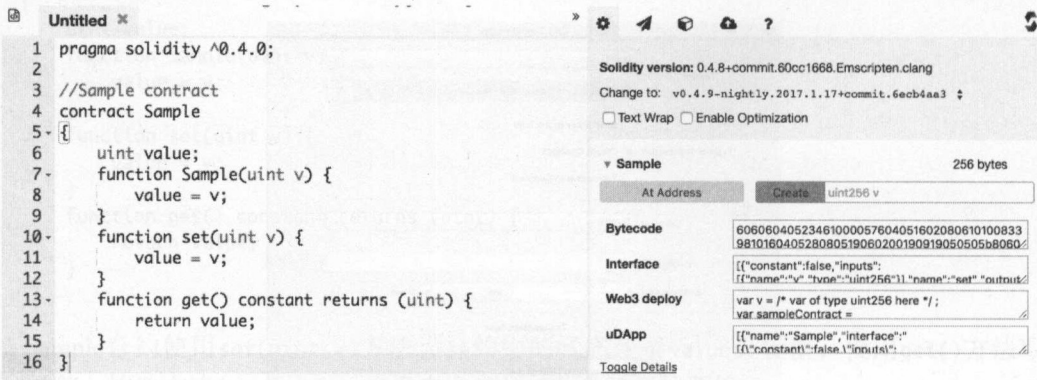


图3.6 在线实时编译器编译合约

^① 地址为<https://ethereum.github.io/browser-solidity/>。

● 获得字节码和ABI文件

bytecode是合约的字节码文件,也就是能真正被EVM运行的文件,Sample合约的字节码如下:

Interface中的字符串是合约的应用二进制接口，也就是提供给外界访问的接口。合约中的每一个方法（除internal修饰的方法外）都会在ABI中被描述：constant字段表示该方法是否是一个用constant修饰的方法，constant方法不会改变合约状态；inputs表示方法的输入参数和类型；name是方法名；outputs表示方法的输出参数和类型，也即返回值；type是该接口的类型；function表示普通的方法；constructor是构造方法。Sample的ABI如下所示：

Web3 deploy中是JavaScript代码，使用web3.js接口来创建一个合约实例。传入参数包括上面生成的字节码和ABI文件。返回参数包括合约实例、合约地址、交易哈希等。可以使用该合约实例来调用合约中的方法，实现与合约的交互。

```
var v = /* var of type uint256 here */ ;
var sampleContract =
```

```
web3.eth.contract([{"constant":false,"inputs":[{"name":"v","type":"uint256"}],"name":"set",
"outputs":[],"payable":false,"type":"function"}, {"constant":true,"inputs":[],"name":"get",
"outputs":[{"name":"","type":"uint256"}],"payable":false,"type":"function"}, {"inputs":[{"name":
"v","type":"uint256"}],"payable":false,"type":"constructor"}}]);
var sample = sampleContract.new(
    v,
    {
        from: web3.eth.accounts[0],
        data:
'60606040523461000057604051602080610100833981016040528080519060200190919050505b80600081905550
5b505b60c28061003e6000396000f30060606040526000357c010000000000000000000000000000000000000000
0000000000000000900463ffffffff16806360fe47b11460465780636d4ce63c146060575b6000565b34600057605e
60048080359060200190919050506080565b005b34600057606a608b565b604051808281526020019150506040518
0910390f35b806000819055505b50565b600060005490505b905600a165627a7a72305820e737306ae0d72d653875
a62df87529e66cee8e9a30671378e3182f89a27a381ac0029',
        gas: 3000000
    }, function(e, contract){
        console.log(e, contract);
        if (typeof contract.address != 'undefined') {
            console.log('Contract mined! address: ' + contract.address + ' transactionHash: ' +
                contract.transactionHash);
        }
    })
```

uDApp中是编译完成后返回的JSON值，JSON中包含了合约名、字节码和ABI。如果使用web3.js接口编译Sample合约，返回的就是该JSON值。

```
[
  {
    "name": "Sample",
    "interface": [{"constant": false, "inputs": [{"name": "v", "type": "uint256"}], "name": "set",
, "outputs": [], "payable": false, "type": "function"}, {"constant": true, "inputs": [], "name": "get", "o
utputs": [{"name": "", "type": "uint256"}], "payable": false, "type": "function"}, {"inputs": [{"name":
"v", "type": "uint256"}], "payable": false, "type": "constructor"}],
    "bytecode": "6060604052346100005760405160208061010083398101604052808051906020019091905
0505b8060000819055505b505b60c28061003e6000396000f30060606040526000357c0100000000000000000000
0000000000000000000000000000000000463ffffffffff16806360fe47b11460465780636d4ce63c146060575b6
000565b34600057605e60048080359060200190919050506080565b005b34600057606a608b565b60405180828152
60200191505060405180910390f35b806000819055505b50565b600060005490505b905600a165627a7a72305820e
737306ae0d72d653875a62f87529e66cee8e9a30671378e3182f89a27a381ac0029"
  }
]
```

● 合约方法测试

如图3.8所示，点击右侧的Create按钮可以在线创建一个Sample合约实例，如果合约中实现了带参数的构造方法，可以在Create的时候传入。

调用set()方法时，在输入框中填入value值，然后点击“set”按钮设置value。点击“get”方法按钮就可以返回value值。在线编译器可以非常方便地可视化测试智能合约中的方法。其数据都存储在内存中，不需要开启以太坊私链。

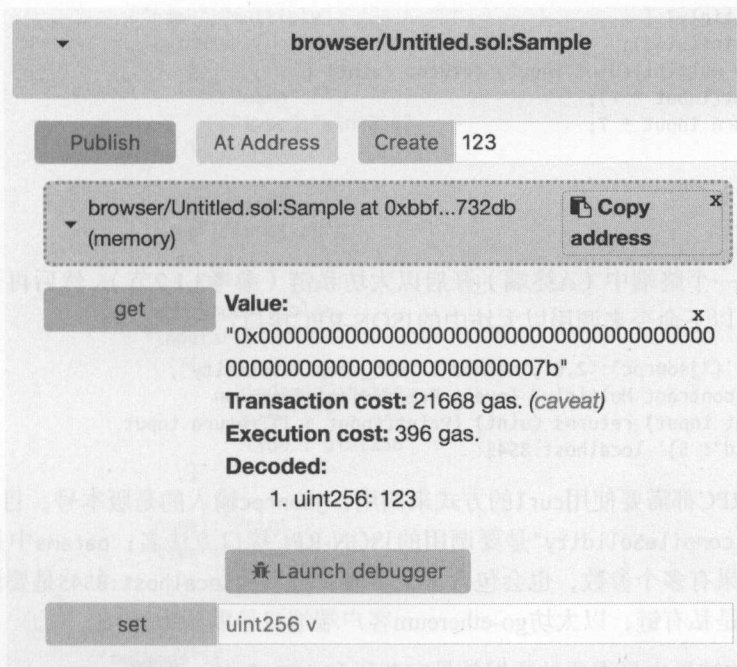


图3.8 合约方法测试

3.3 以太坊编程接口

以太坊区块链平台作为独立的底层平台,需要与外界交互,则必须为外界提供接口,目前RPC接口是以太坊原生支持的,不限语言,跨平台;JavaScript API (web3.js)则是JavaScript对RPC的封装,使用较为简单方便,但是仅限于JavaScript语言调用。

3.3.1 JSON RPC

JSON是一种轻量级的数据交换格式,可以用来表示数字、字符串、值的有序序列和键值对集合。JSON-RPC是一种无状态的、轻量级的远程过程访问协议,并使用JSON (RFC 4627) 作为其数据格式。JSON-RPC主要是在处理过程中定义了一些数据结构和规则,可以在不同的消息传递环境中如Sockets、HTTP中传递信息。

这里使用JSON RPC接口在以太坊私链中部署调用合约。

● 智能合约

下面我们以一个非常简单的合约Multiply7为例进行讲解,该合约有一个multiply()方法,传入一个uint类型数据,乘以7后返回结果。


```
contract Multiply7 {
    event Print(uint);
    function multiply(uint input) returns (uint) {
        Print(input * 7);
        return input * 7;
    }
}
```

• 编译合约

首先需要在—个终端中（A终端）开启以太坊私链（参考3.1.2节）；然后再开启另一个终端（B终端）并输入以下命令来调用以太坊中的JSON-RPC接口：

```
curl --data '{"jsonrpc":"2.0","method": "eth_compileSolidity",
"params": ["contract Multiply7 {event Print(uint);function
multiply(uint input) returns (uint) {Print(input * 7);return input
* 7;}}"], "id": 5}' localhost:8545
```

所有JSON-RPC都需要使用curl的方式来调用，jsonrpc输入的是版本号，目前—般使用2.0；method中的“eth_compileSolidity”是要调用的JSON-RPC接口方法名；params中是调用方法需要传入的参数，如果有多个参数，也会包含多个字段；最后的localhost:8545是要调用的以太坊地址，由于开启的是私有链，以太坊go-ethereum客户端端口号默认为8545。

B终端返回的结果中最重要的数据就是字节码和abiDefinition数据。

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "result": {
    "Multiply7": {
      "code": "0x606060405260788060106000396000f3606060405260e060020a6000350463c6888fa18114601c575b6
002565b3460025760666004356040805160078302815290516000917f24abdb5865df5079dcc5ac590ff6f01d5c16
edbc5fab4e195d9febd1114503da919081900360200190a15060070290565b6040805191825251908190036020019
0f3",
      "info": {
        "source": "contract Multiply7 {event
Print(uint);function multiply(uint input) returns (uint)
{Print(input * 7);return input * 7;}}",
        "language": "Solidity",
        "languageVersion": "0.4.4",
        "compilerVersion": "0.4.4",
        "compilerOptions": "--combined-json bin,abi,userdoc,devdoc --add-std --optimize",
        "abiDefinition": [
          {
            "constant": false,
            "inputs": [
              {
                "name": "input",
                "type": "uint256"
              }
            ],

```

● 获得调用账户

```
curl --data '{"jsonrpc":"2.0","method":"eth_coinbase", "id":1}' localhost:8545
```

```
{"jsonrpc":"2.0","id":1,"result":"0x90c2323cdef75fd82e65ac496fc45eafadf4563"}
```

以太坊中发起交易者要有足够的以太币才能让交易被确认，然后加入到区块链中。在私链中

进行挖矿会把以太币发到coinbase，也就是当前以太坊账户中默认的第一个账户。当然，coinbase可以根据需要进行更改。在B终端中执行以下命令来查看是否有足够的以太币。

```
curl --data '{"jsonrpc": "2.0", "method": "eth_getBalance", "params": ["0x90c2323cdeff75fd82e65ac496fc45eafad4563", "latest"], "id": 2}' localhost:8545
```

返回结果如下，result就是当前账户以太币的数量。

```
{"jsonrpc":"2.0","id":2,"result":"0x27b41fd348ff2fc32d0"}
```

● 部署合约

在B终端中执行以下命令，`params`参数中的`from`就是发起该笔交易的账户，`data`参数中就是该合约的字节码。

```
curl --data '{"jsonrpc": "2.0", "method": "eth_sendTransaction",  
"params": [{"from":  
"0x90c2323cdeff75fd82e65ac496fc45eafad4563", "gas": "0xb8a9",  
"data":  
"0x6060604052605f8060106000396000f3606060405260e060020a6000350463c6888fa18114601a575b005b6058  
6004356007810260609081526000907f24abdb5865df5079dcc5ac590ff6f01d5c16  
edbc5fab4e195d9febd1114503da90602090a15060070290565b5060206060f3"}], "id": 6}'  
localhost:8545
```

在B终端执行上述命令后，交易需要被挖矿才能确认，所以在A终端中执行以下命令启动挖矿。

```
miner.start()
```

同时会显示出当前合约部署成功后的合约地址，使用该合约地址才能调用合约中的方法。`created`字段就是合约地址。

Tx(0x183685e0ac93e80b0cde97e8b53d425e15bf84d76b073586ad23c66e53085f1f) created:
0x1269e13d9df9f64b9cbf0bdfaf0dcb0e67f4cf06

B终端返回结果如下，其中result是本次交易（部署合约）的交易哈希。

```
{ "jsonrpc": "2.0", "id": 6, "result": "0x183685e0ac93e80b0cde97e8b53d425e15bf84d76b073586ad23c66e53085f1f" }
```

● 调用合约方法

调用合约方法如下，其中to参数中就是合约地址，data就是调用合约需要传入的参数。data参数中包括了调用合约中的哪个方法和具体参数值，可以简称为payload。同时确认该交易也需要在A终端挖矿。

[illegible]

在A终端计算payload中的方法选择符对应的字节，选取Keccak哈希表的前4个字节，并进行十六进制编码。

```
> web3.sha3("multiply(uint256)").substring(0, 8)
"0xc6888f"
```

假设要传入的值为6，是一个uint256类型，将会被编码成：

```
0000000000000000000000000000000000000000000000000000000000000006
```

将方法选择符和编码参数结合起来，就生成了上述data中的数据。

B终端返回结果如下，通过返回result中的交易哈希可以查询本次交易的详情。

```
{"jsonrpc": "2.0", "id": 8, "result": "0xf3efaa16da4c439eb960387e35468a6ac527f5d9586ebe7062f69c319364661b"}
```

● 根据交易哈希查找交易详情

一笔交易发生后，真正的返回结果只是交易哈希。可以利用该交易哈希获得本次交易的真正详情以及一些方法的返回值。在B终端中执行以下命令获取交易的结果和返回值。

```
curl --data '{ "jsonrpc": "2.0", "method": "eth_getTransactionReceipt", "params": [ "0xf3efaa16da4c439eb960387e35468a6ac527f5d9586ebe7062f69c319364661b" ], "id": 1 }' localhost:8545
```

B终端返回结果如下，其中logs.data字段就是方法真正的返回值。把十六进制的2a转换成十进制就是42，符合预期，合约方法调用通过。

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "blockHash": "0x9c20aefe176d8c1ef309411ed943543286800602053f40c9cb610506c90eec32",
    "blockNumber": "0x2269",
    "contractAddress": null,
    "cumulativeGasUsed": "0x11110",
    "from": "0x90c2323cdeff75fd82e65ac496fc45eafadf4563",
    "gasUsed": "0x5867",
    "logs": [
      {
        "address": "0x1269e13d9df9f64b9cbf0bdfaf0dc0e67f4cf06",
        "topics": [
          "0x24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"
        ],
        "data": "0x000000000000000000000000000000000000000000000000000000000000002a",
        "blockNumber": "0x2269",
        "transactionIndex": "0x1",
        "transactionHash": "0xf3efaa16da4c439eb960387e35468a6ac527f5d9586ebe7062f69c319364661b",
        "blockHash": "0x9c20aefe176d8c1ef309411ed943543286800602053f40c9cb610506c90eec32",
        "logIndex": "0x0"
      }
    ]
  },
}
```



```

compilerOptions: "--combined-json bin,abi,userdoc,devdoc
--add-std --optimize",
  compilerVersion: "0.4.4",
  developerDoc: {
    methods: {}
  },
  language: "Solidity",
  languageVersion: "0.4.4",
source: "contract Multiply7 {event Print(uint);function
multiply(uint input) returns (uint) {Print(input * 7);return input * 7;}}",
  userDoc: {
    methods: {}
  }
}
}
}

```

分别把字节码和ABI赋值给两个变量，方便后续使用：

```

> var code = compiled.Multiply7.code;
undefined
> var abi = compiled.Multiply7.info.abiDefinition
undefined

```

查看code和ABI变量：

```

> code
"0x606060405260788060106000396000f3606060405260e060020a6000350463c6888fa18114601c575b6002565b
3460025760666004356040805160078302815290516000917f24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fa
b4e195d9febd1114503da919081900360200190a15060070290565b60408051918252519081900360200190f3"
> abi
[[
  constant: false,
  inputs: [{
    name: "input",
    type: "uint256"
  }],
  name: "multiply",
  outputs: [{
    name: "",
    type: "uint256"
  }],
  payable: false,
  type: "function"
], {
  anonymous: false,
  inputs: [{
    indexed: false,
    name: "",
    type: "uint256"
  }],
  name: "Print",
  type: "event"
}]

```

● 部署合约

可以使用上述生成的code和ABI把合约部署到以太坊上,同时需要挖矿来确认该笔交易。可以看到合约已经部署,合约地址为0xa69c52ca29e042e63c42bb46278d4c44fd0fd835。

```
> web3.eth.contract(abi).new({from:
  "0x90c2323cdeff75fd82e65ac496fc45eafadf4563", data: code})
I0119 12:25:13.203287 internal/ethapi/api.go:1045]
Tx(0x7c3959992f31cb7a828d0805c8e78b52a91cb47475eb9f8663d42677a7a9529c) created:
0xa69c52ca29e042e63c42bb46278d4c44fd0fd835
{
  abi: [{
    constant: false,
    inputs: [{...}],
    name: "multiply",
    outputs: [{...}],
    payable: false,
    type: "function"
  }, {
    anonymous: false,
    inputs: [{...}],
    name: "Print",
    type: "event"
  }],
  address: undefined,
  transactionHash: "0x7c3959992f31cb7a828d0805c8e78b52a91cb47475eb9f8663d42677a7a9529c"
}
```

● 调用合约方法

合约成功部署后可以获得合约地址,使用该合约的ABI和合约地址可以创建一个合约实例,使用创建的合约实例可以调用合约中的方法,返回的var multi就是合约实例:

```
> var multi =
web3.eth.contract(abi).at("0xa69c52ca29e042e63c42bb46278d4c44fd0fd835")
```

然后使用上一步中的合约实例multi来调用合约中的multiply()方法,传入的参数为6。这一步同样需要挖矿进行确认:

```
> multi.multiply.sendTransaction(6, {from:
  "0x90c2323cdeff75fd82e65ac496fc45eafadf4563"})
I0119 12:33:05.872576 internal/ethapi/api.go:1047]
Tx(0x75142a971e69494bd0729d9aea969b4631cf1db8c77f5855e4714e6149c72a7a) to:
0xa69c52ca29e042e63c42bb46278d4c44fd0fd835
"0x75142a971e69494bd0729d9aea969b4631cf1db8c77f5855e4714e6149c72a7a"
```

由于使用web3调用合约中的方法无法返回真正的计算值,返回的只是交易哈希,所以可以通过合约中定义的Print事件来获得本次交易的详情:

```
> multi.Print(function(err, data)
{ console.log(JSON.stringify(data)) })
```

返回交易详情如下所示，args中的42就是通过合约方法计算后需要返回的真正的值，符合预期，合约方法调用成功。

```
{
  "address": "0xa69c52ca29e042e63c42bb46278d4c44fd0fd835",
  "args": {
    "": "42"
  },
  "blockHash": "0x2f207f6107c203638761915d175bf52e3958b057e547fc384ba4d65cb7b74410",
  "blockNumber": 8962,
  "event": "Print",
  "logIndex": 0,
  "removed": false,
  "transactionHash": "0x75142a971e69494bd0729d9aea969b4631cf1db8c77f5855e4714e6149c72a7a ",
  "transactionIndex": 0
}
```

3

3.4 DApp 开发框架与流程

基于现有的开发框架来开发实际的项目可大大加快项目开发进度，在以太坊去中心化应用的开发中，目前比较常用的开发框架有Meteor和Truffle。本节给出了一种分层可扩展的开发流程。开发者可根据项目的类型、规模、难度等选择合适的开发框架与流程。

3.4.1 Meteor

Meteor是一套通用的webapp前端开发框架，可以非常方便地集成以太坊的web3.js接口。Meteor被认为是一个全栈式的框架，完全使用JavaScript实现，并提供了重新加载、CSS注入和支持预编译（Less、CoffeeScript等）。Meteor可以非常方便地构建一个单页面应用（Single Page App，SPA），可以把所有的前端代码都写入到index.html中，并使用一个js文件和css文件加入在资源中。Meteor支持响应式的开发，类似于Angular.js，可以非常简单地构建界面。

目前有很多DApp应用都是基于Meteor框架开发的，下面介绍如何安装Meteor、加载以太坊web3模块、调用web3.js接口以及部署DApp应用。

• 安装Meteor

Meteor的官方下载地址为<https://www.meteor.com/install>。不同的操作系统有不同的下载方式，Windows需要下载安装包，macOS和Linux可以直接使用如下命令行在终端中下载：

```
curl https://install.meteor.com/ | sh
```

安装过程较为缓慢，安装完成后，在终端中输入以下命令：

```
meteor --version
```

如果可以成功显示Meteor版本号，则表示Meteor成功安装。

● 加载以太坊web3模块

Meteor安装成功后就可以用来开发DApp了。首先在命令行执行以下命令创建一个Meteor项目：

```
meteor create 项目名
```

该过程可能较慢，因为需要加载多个初始模块。然后执行以下命令进入项目目录：

```
cd 项目名
```

执行以下命令加载web3：

```
meteor add ethereum:web3
```

如果由于版本原因导致上述加载失败，可以使用下面的命令代替：

```
meteor npm install web3 --save
```

因为以太坊的web3.js接口使用更为简单方便，也已经较为成熟，所以推荐在Meteor中集成web3接口来开发DApp。

● 调用以太坊web3.js接口

Meteor项目创建完成后，默认会有两个文件夹client和server，分别对应的是客户端程序和服务端程序。根据实际需要，可以分别在客户端和服务端使用web3.js。

在client文件夹中新建lib文件夹，在lib文件夹中新建init.js，在init.js中实现如下初始化web3代码：

```
var Web3 = require('web3');
if (typeof web3 !== 'undefined') {
    web3 = new Web3(web3.currentProvider);
}
else {
    //连接以太坊客户端，如在本机上运行的以太坊私有链，默认端口8545
    web3 = new Web3(new
    Web3.providers.HttpProvider("http://localhost:8545"));
}
console.log("client:" + web3.eth.accounts[0]);
```

在客户端初始化web3实例后，就可以调用web3.js接口中所有的方法了，一个DApp框架即可搭建起来了。当客户端在浏览器中被加载后，就会在Console面板中打印出当前以太坊的第一个账号。

同样可以在server/main.js中实现以上代码，这样就能在服务端调用web3.js中的所有接口，进而实现与智能合约的交互。

● 部署DApp应用

在部署DApp应用之前，首先需要在另一个终端中开启以太坊私有链。然后在项目目录中执行meteor命令即可自动化部署应用。成功部署DApp的命令行如下所示，其中打印出的

“server:0x90c2323cdeff75fd82e65ac496fc45eafadf4563”为在server/main.js中实现的代码，打印的地址为当前以太坊私链中的第一个账户。

```
→ meteor
[[[[[ ~/Desktop/myapp ]]]]]
```

```
=> Started proxy.
```

```
=> Started MongoDB.
```

```
I20170608-23:54:53.293(8)? server:0x90c2323cdeff75fd82e65ac496fc45eafadf4563
```

```
=> Started your app.
```

```
=> App running at: http://localhost:3000/
```

用Meteor开发的DApp默认使用的端口为3000端口，可以在浏览器中输入http://localhost:3000来访问DApp应用，如图3.9所示。

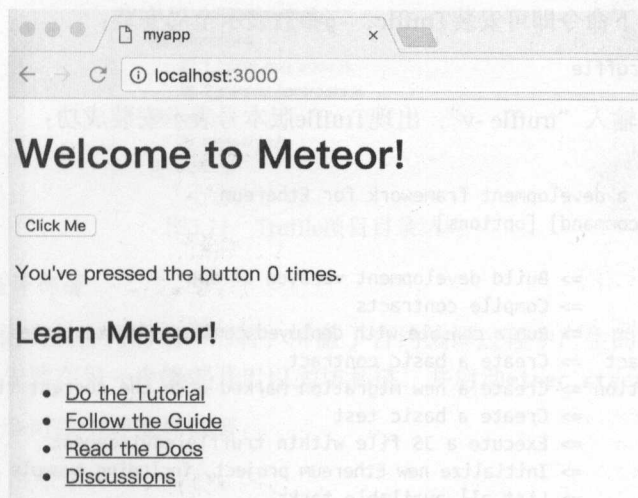


图3.9 启动Meteor应用

打开浏览器的开发者选项，切换到Console控制台页面，可以看到打印出的信息，该打印信息就是在client/lib/init.js中实现的代码，获得当前以太坊中的第一个账户，如图3.10所示。在命令行中也可以交互式地调用web3.js。这样就能使用Meteor来开发一个简单的DApp。

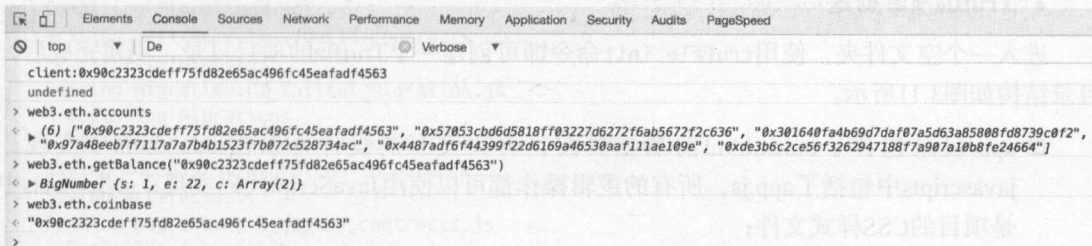


图3.10 Meteor中使用web3.js接口

3.4.2 Truffle

Truffle是一款非常优秀的开发DApp的框架。在Truffle中可以方便地使用JavaScript进行应用的开发,并使用JavaScript中几乎所有的机制,如Promise、异步调用等。Truffle使用了包装web3.js的一个Promise框架Pudding,所以不需要手动加载web3.js库,可以有效提升开发效率。同时Truffle也内置了智能合约编译器,只要使用脚本命令就可以完成合约的编译、动态库链接、部署、测试等工作,大大简化了合约的开发生命周期。

下面介绍Truffle的安装、项目创建及客户端应用的运行,并使用Truffle的一个默认合约MetaCoin来演示智能合约的编译和部署。

● Truffle安装

在终端中执行以下命令即可安装Truffle, -g参数表示全局安装:

```
npm install -g truffle
```

安装后在终端中输入“truffle -v”,出现Truffle版本号表示安装成功:

```
→ truffle -v
Truffle v2.1.1 - a development framework for Ethereum
Usage: truffle [command] [options]
Commands:
  build           => Build development version of app
  compile         => Compile contracts
  console         => Run a console with deployed contracts instantiated and available (REPL)
  create:contract => Create a basic contract
  create:migration => Create a new migration marked with the current timestamp
  create:test     => Create a basic test
  exec           => Execute a JS file within truffle environment
  init           => Initialize new Ethereum project, including example contracts and tests
  list           => List all available tasks
  migrate         => Run migrations
  networks        => Show addresses for deployed contracts on each network
  serve          => Serve app on localhost and rebuild changes as needed
  test           => Run tests
  version        => Show version number and exit
  watch          => Watch filesystem for changes and rebuild the project automatically
```

● Truffle项目创建

进入一个空文件夹,使用truffle init命令即可创建一个Truffle的项目目录,创建完成后,目录结构如图3.11所示。

- ❑ app/目录包含了index.html前端主页面,有关交互的设计可以在index.html中实现; javascripts中包括了app.js,所有的逻辑操作都可以使用JavaScript代码实现; stylesheets中是项目的CSS样式文件;
- ❑ contracts/目录包含了所有的智能合约,默认已经创建了3个合约;

- ❑ migrations/目录是有关合约部署的配置文件, 如果新增加合约, 需要进入2_deploy_contract进行配置;
- ❑ test/目录是合约测试代码, 可以实现对合约的单元测试;
- ❑ truffle.js是整个DApp项目的配置文件, 包括host、端口号的配置。

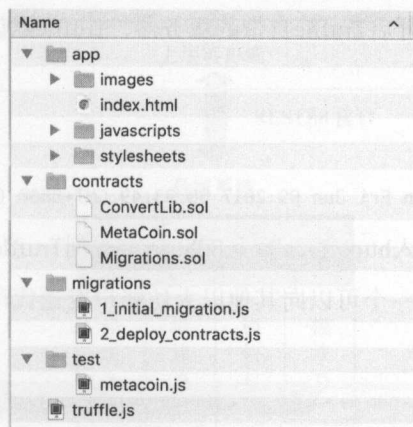


图3.11 Truffle项目目录结构

● 智能合约的编译部署

Truffle可以自动化实现合约的编译部署, 屏蔽了合约编译过程中产生的字节码和ABI文件。在部署合约之前, 首先要在另一个终端开启以太坊私链, 并启动`miner.start()`挖矿。

分别使用以下命令可以实现编译部署:

```
truffle compile
truffle migrate
```

成功部署合约后的界面如下所示, 其中Migrations、ConvertLib、MetaCoin这3个合约后的十六进制数就是合约地址。

```
→ truffle compile
Compiling ConvertLib.sol...
Compiling MetaCoin.sol...
Compiling Migrations.sol...
Writing artifacts to ./build/contracts
→ truffle migrate
Running migration: 1_initial_migration.js
  Deploying Migrations...
  Migrations: 0x4fdc14e0a97e9c1a5a1fce10e7bd9675f29ebb8a
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying ConvertLib...
  ConvertLib: 0x4be659e79f4c3a98b721383cdb3aaa12e659da3d
Linking ConvertLib to MetaCoin
```



```
Deploying MetaCoin...
MetaCoin: 0xac824e61ee2b56c81a3d63447c5dd2a6f95499aa
Saving successful migration to network...
Saving artifacts...
```

• 客户端应用运行

合约成功部署后, 就可以开启Truffle自带的服务器, Truffle默认使用8080端口, 使用“truffle serve”即可开启服务器:

```
→ truffle serve
Serving app on port 8080...
Rebuilding...
Completed without errors on Fri Jun 09 2017 00:03:49 GMT+0800 (CST)
```

此时可以打开浏览器, 输入http://localhost:8080即可看到Truffle应用了, 如图3.12所示。

示例中的MetaCoin模拟了一个可以向其他以太坊账户发送代币的功能。我们将在3.5节详细讲解MetaCoin应用并进行优化。



3.12 启动Truffle应用

本书第8章中的通用积分系统采用的就是Truffle框架开发的去中心化应用。

3.4.3 分层可扩展开发流程

使用3.4.1节与3.4.2节中介绍的框架来开发有网页的去中心化应用非常方便, 架构相对比较简单, 同时可以快速部署到服务器上。但是Truffle等框架的使用有较大的局限性, 最大的一个问题

就是开发的系统很难实现跨平台。如果在实际的去中心化商业项目中,要求客户端同时运行在浏览器、移动客户端和其他终端上,则Truffle就明显不能满足需求。为了解决这个问题,需要设计一套分层可扩展的项目开发流程,这种模式在现在的行业开发过程中的运用已经很普遍了。

可扩展项目开发流程的大致架构如图3.13所示。

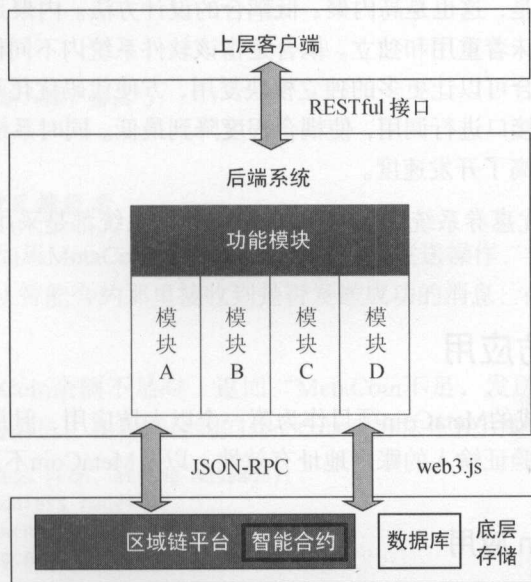


图3.13 分层可扩展架构

该架构主要分为三层,下面我们分别进行介绍。

(1) 底层存储。底层使用区块链保证交易的安全性和不可篡改性。平台可以是以太坊Ethereum、超级账本Hyperledger、趣链Hyperchain等主流的区块链平台。核心的业务逻辑全部使用智能合约实现,并把智能合约部署到区块链上运行。在底层存储中使用数据库是为了对区块链数据做一个完整的备份,实现灾备。同时因为目前行业内还没有很好的区块数据可视化的解决方案,所以加入数据库可以间接查看区块链的数据。在实际开发中,可以进行区块链和数据库的双写,以实现数据的同步。

(2) 后端系统。中间层作为上层应用和底层区块链的桥接。后端系统可以使用多种不同的接口和区块链进行交互,常见的有JSON-RPC接口和web3.js接口。如果使用Java作为后端,则可以选择JSON-RPC;如果使用Node.js作为后端,则可以选择web3.js。关于接口的选型,可以参考3.3节。由于关键的合约逻辑都已经在智能合约中实现,所以后台实现的主要功能就是数据的编码、解码和转发,以及为上层提供RESTful接口调用。

(3) 上层客户端。客户端直接面向用户提供服务。客户端可以广义理解为浏览器网页、PC客户端、移动客户端等。这些客户端都可以使用中间层提供的RESTful接口来和区块链实现交互。

中间层服务和底层区块链对用户是透明的，并且是一种轻客户端的设计，不需要把太多和区块链相关的加密解密、解码编码等复杂操作放在客户端来实现，客户端一般只需要处理RESTful返回的JSON数据。

使用这种分层的可扩展项目开发流程符合软件工程的思想，不同的层次可以让不同的开发人员进行设计实现。重要的是，这也是高内聚、低耦合的设计方法。内聚是指一个模块内各个元素彼此结合的紧密程度，意味着重用和独立。耦合是指该软件系统内不同模块之间的互联程度。设计原则符合高内聚、低耦合可以让更多的独立模块复用，方便代码优化和扩展。采用分层可扩展的模式，不同平台仅使用接口进行调用，使耦合程度降到最低。同时系统在开发中可以进行多人并行开发，对接容易，提高了开发速度。

本书第8章中的电子优惠券系统和第9章中的数字票据系统都是采用分层可扩展的开发流程开发的。

3.5 第一个以太坊应用

这里把Truffle默认生成的MetaCoin项目作为第一个以太坊应用，但是MetaCoin还有不完善的地方，需要优化，比如，验证输入的账户地址有效性，以及MetaCoin不足时的反馈提示。

3.5.1 优化 MetaCoin 应用

- 验证账户地址有效性

在原有的MetaCoin项目中，可以在输入地址栏中输入任何字符，然后向这些非法地址发送MetaCoin。这些地址不是正确的以太坊账户，不符合实际的需要。在发送MetaCoin时，需要对目的账户进行验证，检测其是否是当前以太坊客户端中已经存在的账户地址，验证通过才能执行调用合约方法，否则给出提示消息。

在app.js文件中实现判断账户是否存在的方法是isAccountCorrect()。

```
function isAccountCorrect(receiver) {
    for(var i = 0; i < accounts.length; i++) {
        if(receiver == accounts[i]) {
            return true;
        }
    }
    return false;
}
```

同时在sendCoin()方法中做如下判断：

```
function sendCoin() {
    var meta = MetaCoin.deployed();
    var amount = parseInt(document.getElementById("amount").value);
    var receiver = document.getElementById("receiver").value;
```

```

//判断账号有效性
if (isAccountCorrect(receiver)) {
    setStatus("输入账号正确");
    meta.sendCoin(receiver, amount, {from: account, gas:
        1000000}).then(function (result) {
        ...
    }).catch(function (e) {
        ...
    });
} else {
    setStatus("输入账户错误");
}
}

```

• MetaCoin不足时反馈提示

在原有的应用中，如果MetaCoin余额不足，继续执行发送操作，仍然提示“send complete”发送完成操作，并没有从智能合约那里接收到是否发送成功的消息，在使用中会有误导，所以需要优化。

在合约中，当MetaCoin余额不足时，返回“MetaCoin不足，发送失败”的消息，否则返回“MetaCoin发送成功”的消息。在智能合约的sendCoin()方法中使用Transfer()事件返回消息。

```

event Transfer(address from, string message);
function sendCoin(address receiver,
    uint amount)returns(bool) {
    if (balances[msg.sender] < amount) {
        //发送失败
        ...
        Transfer(msg.sender, "MetaCoin不足，发送失败");
        return false;
    } else {
        //发送成功
        ...
        Transfer(msg.sender, "MetaCoin发送成功");
        return true;
    }
}

```

在app.js文件中，当合约实例调用合约事件Transfer()时，获得从合约返回的数据：

```

meta.sendCoin(receiver, amount, {from: account, gas:
    1000000}).then(function (result) {
    var eventTransfer = meta.Transfer();
    eventTransfer.watch(function (error, event) {
        //event内包括了从合约返回的所有数据，event.args.message即可获得
        //发送成功或失败的消息
        ...
    });
}).catch(function (e) {
    ...
});

```


3.5.2 MetaCoin 代码详解

下面对MetaCoin应用的代码结构和代码实现进行进一步介绍, 主要包括合约代码、前端HTML、JavaScript接口调用以及开发中的注意点。

• index.html实现

index.html是整个应用的主界面, 由于目前是单页面应用, 所有的界面元素都实现在index.html中。index.html需要加载JavaScript代码, 也就是应用中的app.js, 所以需要在index.html的<head>标签中进行如下实现:

```
<head>
<title>第一个以太坊应用-MetaCoin</title>
<link href='https://fonts.googleapis.com/css?family=Open+Sans:400,700'
rel='stylesheet' type='text/css'>
<link href='./app.css' rel='stylesheet' type='text/css'>
<script src='./app.js'></script>
</head>
```

<title>标签是浏览器中显示的该页面的标题, <link>标签是加载css文件, <script>标签是加载JavaScript脚本文件即app.js。

• 合约代码实现

由于MetaCoin合约较为简单, 这里做初步的介绍。MetaCoin合约完整的实现如下。

mapping类似于哈希, 是一种键值对的数据结构, “=>”左侧为输入的键的类型, 右侧为查找的值的类型。这里使用address类型的账户地址去查找uint类型的MetaCoin余额。

MetaCoin()为构造方法, 在构造方法中初始化交易发起者的初始余额为10000。

sendCoin()为发送MetaCoin的方法, 当余额不足时将发送失败。合约中有两种类型的方法: 交易方法和constant方法。交易方法是会改变以太坊状态的方法, 也就是会修改状态变量的方法, 执行这种方法也称为执行一笔交易, 需要消耗gas, 这里的sendCoin()方法就是一个交易方法。当使用web3来调用交易方法时, 是无法获得该方法真正的返回值的, 例如sendCoin()方法返回的bool值其实在web3接口调用中是无法获取的, 真正返回的值是一个32字节的交易哈希。但是在合约内部方法调用时是可以获取交易方法的返回值的, 所以这里可以使用event事件向web3返回数据, 比如“发送失败”的字符串。

getBalance()方法就是一个constant方法, 只用来获取以太坊数据和变量的值, 不会导致以太坊状态的改变。可以手动修饰一个方法为constant, 如果方法没有constant修饰, 以太坊也会自动识别哪些方法是constant方法。对于constant方法, 可以使用web3接口成功接收返回值, 如getBalance()方法返回的uint类型的余额, 可以在app.js中接收到。

```
contract MetaCoin {
    //根据账户地址查找账户余额
    mapping (address => uint) balances;
```

```

//初始账户余额为10000
function MetaCoin() {
    balances[tx.origin] = 10000;
}
//event事件向web3返回数据
event Transfer(address from, string message);
function sendCoin(address receiver,
    uint amount)returns(bool) {
    if (balances[msg.sender] < amount) {
        //发送失败
        Transfer(msg.sender, "MetaCoin不足, 发送失败");
        return false;
    }
    else {
        //发送成功
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        Transfer(msg.sender, "MetaCoin发送成功");
        return true;
    }
}
//获得账户MetaCoin
function getBalance(address addr)constant returns(uint) {
    return balances[addr];
}
}

```

• app.js实现

app.js中处理所有的业务逻辑,所有的web3接口方法在这里调用。app.js相当于是一个中间层,接收前端页面的输入,并调用合约方法处理,然后接收合约方法的处理结果,并显示在页面上。

在页面被加载时,JavaScript会执行window.onload()方法,web3.eth.getAccounts()方法即调用了web3接口,用来获取当前以太坊客户端(私链)中的所有账户。这里设置account变量为账户数组中的第一个账户,也是后续发起合约方法调用的默认账户。

```

window.onload = function() {
    web3.eth.getAccounts(function(err, accs) {
        if (err != null) {
            alert("获取以太坊账户失败");
            return;
        }
        if (accs.length == 0) {
            alert("以太坊账户为空");
            return;
        }
        accounts = accs;
        account = accounts[0];
        refreshBalance();
    });
};

```

refreshBalance()用来刷新界面上显示的余额值。MetaCoin.deployed()可以非常方便地获取已经部署的MetaCoin合约的实例,使用合约实例可以方便地调用合约方法。constant方法可以使

用call()方法调用,也可以默认省略call()。from为发起交易的账户,该笔交易会使用该账户的私钥进行加密,在constant方法中,from参数可以省略,省略的from使用以太坊第一个账户作为默认值。在回调函数中可以接收到constant方法的返回值。

```
function refreshBalance() {
    var meta = MetaCoin.deployed();
    meta.getBalance.call(account, {from:
account}).then(function(value) {
    var balance_element = document.getElementById("balance");
    balance_element.innerHTML = value.valueOf();
}).catch(function(e) {
    console.log(e);
    setStatus("获得余额失败");
});
}
```

sendCoin()方法用来发送MetaCoin,当调用的是交易方法时,from参数不能省略,必须显式地声明是由哪一个账户发起的。同时还需要一个gas参数,交易方法在以太坊上执行需要消耗一定量的gas,当gas参数不指定时,会发送一个默认的gas值,一般情况下方法可以成功执行。但是当一个合约方法代码量较多时,可能会造成OOG(out of gas)的报错,导致方法调用失败。解决方法就是显式地发送一个较大的gas值。在sendCoin()回调函数中的result中,可以获得本次交易的哈希值。event实例对象使用watch()调用,开启监听,返回的event变量中包含了本次交易的详细信息和返回值。这里使用alert弹出对话框的方式显示出交易方法的返回值和event事件返回值,方便观察。event使用完成后,建议使用stopWatching()停止监听,否则会不断地轮询,造成资源消耗。

```
function sendCoin() {
    var meta = MetaCoin.deployed();
    var amount = parseInt(document.getElementById("amount").value);
    var receiver = document.getElementById("receiver").value;
    if (isAccountCorrect(receiver)) {
        setStatus("发送进行中,请等待...");
        meta.sendCoin(receiver, amount, {from: account, gas:
1000000}).then(function (result) {
            var eventTransfer = meta.Transfer();
            eventTransfer.watch(function (error, event) {
                alert("sendCoin()方法返回: " + result + "\n\n Transfer()事件返回: " +
JSON.stringify(event));
                setStatus(event.args.message);
                refreshBalance();
                eventTransfer.stopWatching();
            });
        }).catch(function (e) {
            console.log(e);
            setStatus("发送MetaCoin出现异常");
        });
    }
    else {
        setStatus("输入账户错误");
    }
}
```

3.5.3 MetaCoin 应用运行

进行优化后，第一个以太坊应用的运行结果如图3.14到图3.18所示。



图3.14 输入的以太坊账户错误

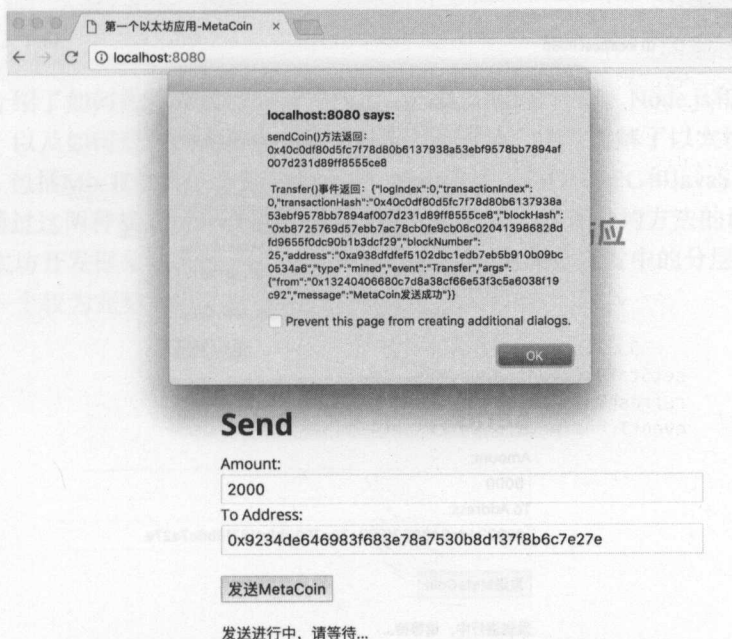


图3.15 发送MetaCoin

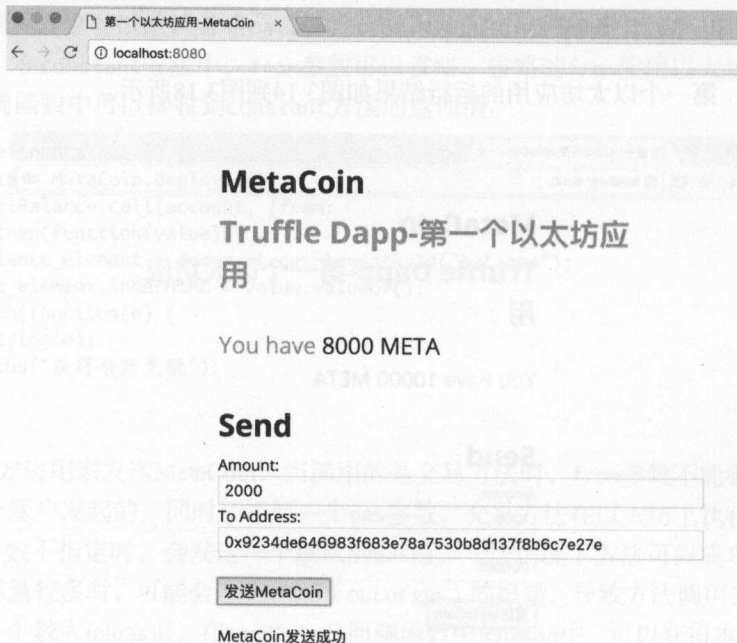


图3.16 成功发送MetaCoin

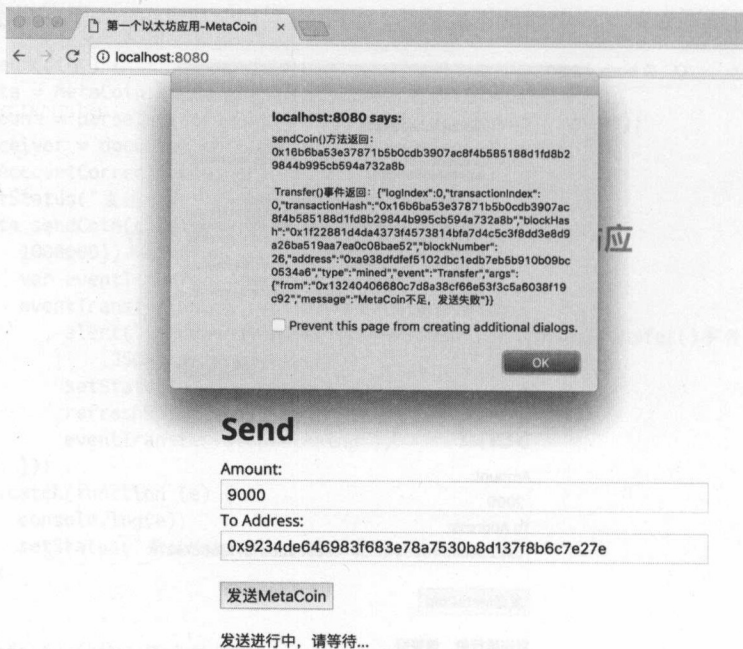


图3.17 发送MetaCoin

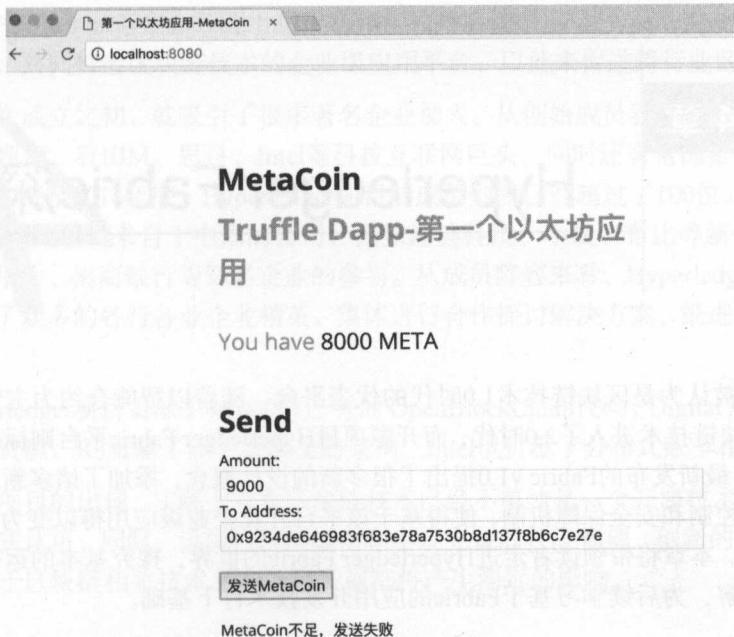


图3.18 发送MetaCoin失败

3.6 本章小结

本章首先介绍了如何搭建以太坊的开发环境，包括Go语言环境、Node.js和npm的配置、solc编译器的安装，以及如何使用以太坊geth客户端搭建私有链；接着讲解了以太坊智能合约开发的集成开发环境，包括Mix IDE和在线实时编译器；然后讲述了JSON RPC和JavaScript API两种以太坊编程接口，通过这两种接口可以实现和以太坊底层的交互，实现合约方法的调用；随后讲述了目前主流的以太坊开发框架与流程，包括Metero、Truffle和商业化开发中的分层可扩展开发流程。最后给出了第一个较为完整的以太坊应用开发实例。

比特币普遍被认为是区块链技术1.0时代的代表平台，随着以智能合约为主要特征的以太坊平台的诞生，区块链技术进入了2.0时代，而开源项目Hyperledger Fabric平台则标志着区块链技术3.0时代的到来。最新发布的Fabric v1.0提出了很多新的设计概念，添加了诸多新的特性，提供了较为完备的权限控制和安全保障机制，使得基于该平台开发企业级应用得以变为现实，平台的关注度也越来越高。本章将带领读者走进Hyperledger Fabric的世界，探究基本的运行原理，从而加深对该平台的了解，为后续学习基于Fabric的应用开发技术打下基础。

4.1 项目介绍

Hyperledger项目是致力于推进区块链数字技术和交易验证的开源项目，目标是让开源社区成员共同合作，共建开放平台，满足来自不同行业的用户需求，并简化业务流程。通过创建分布式账本的公开标准，实现虚拟和数字形式的价值交换。

4.1.1 项目背景

以比特币为代表的“数字加密货币”在区块链技术的支撑下取得了巨大的成功，它们的活跃用户数量和交易量逐年递增，发展程度大大超出了人们的估计。随着比特币等“数字加密货币”成为热点，许多创业者、公司和金融机构渐渐意识到了区块链技术的价值，都认为它可以有更大的应用前景，而不仅仅局限于“数字加密货币”领域。

为此，Vitalik创立了Ethereum项目，希冀打造一个图灵完备的智能合约编程平台，让区块链爱好者可以更好更简单地构建开发区块链应用。继而，市场中涌现出了很多新型区块链应用，比如资产登记、预测市场、身份认证等各类应用。但是，当时的区块链技术自身仍存在着一些无法克服的问题。比如，首先交易效率低下，比特币整个网络只能支持大约每秒7笔左右的交易；其次，对于交易的确定性还无法得到很好的保证；最后，达成共识所采用的挖矿机制会造成很大的资源浪费。这些问题导致了当时的区块链技术无法满足大多数商业应用的需求。

因此，设计并实现一个满足商业需求的区块链平台成为当时区块链发展的一个关键。在社会

各界的强烈呼声中，Linux基金会开源组织于2015年12月启动了名为Hyperledger的开源项目，意在通过各方合作，共同打造区块链技术的企业级应用平台，以此来促进跨行业区块链的发展。

Hyperledger在成立之初，就吸引了很多著名企业加入。从创始成员看，第一批加入的成员几乎都是各行业的翘楚，有IBM、思科、Intel等科技互联网巨头，同时还有富国银行、摩根大通这类金融行业大鳄。截至2016年底，Hyperledger项目的成员列表已经超过了100位。值得一提的是，项目成员中超过1/4的成员来自于中国的公司，比如趣链科技、小蚁、布比等新区区块链公司，同时也有万达、华为、招商银行等知名企业的参与。从成员阵容来看，Hyperledger开源项目声势异常浩大，汇集了众多的各行各业企业精英，集体进行合作探讨解决方案，推进企业级区块链平台的发展。

IBM向Hyperledger项目贡献了44 000行已有的OpenBlockChain代码，Digital Asset则贡献了企业和开发者相关资源，R3贡献了新的金融交易架构，Intel也贡献了分布式账本相关的代码。

Hyperledger项目的出现，实际上宣布区块链技术已经不单纯是一个开源技术了，它已经被主流机构和市场正式认可；同时，该项目首次提出和实现完备的权限管理、创新的一致性算法和可插拔的框架，对于区块链相关技术和产业的发展都将产生深远的影响。

4

4.1.2 项目简介

Hyperledger项目是一个大型的开源项目，希望通过各方合作，共同促进和推进区块链技术在商业应用方面的发展。在组成结构上，包含了很多相关的具体子项目。这些子项目可以是一个独立的项目，也可以是与其它项目关联的项目，比如构建工具、区块链浏览器等。Hyperledger对于子项目的形式并没有给出太大的约束，只要有与之相关的好的想法，都可以向Hyperledger委员会发出申请提案。

项目官方地址托管在Linux基金会网站，代码托管在Gerrit上，并通过GitHub提供代码镜像。为了更好地管理子项目和发展项目，Hyperledger项目成立了一个称为技术指导委员会（Technical Steering Committee, TSC）的机构，这也是Hyperledger项目的最高权力机构，子项目的管理以及整个项目生态的发展等重要决定都将由它执行。Hyperledger项目在管理所属子项目时采用了一种生命周期的形式，赋予每个项目一个生命周期，方便项目的运行和管理。整个生命周期分为5个阶段，分别是提案（proposal）阶段、孵化（incubation）阶段、活跃（active）阶段、弃用（deprecated）阶段以及最后终止（End of Life）阶段。每个项目在开发运行过程中，一个时间点只会对应着一个阶段。当然，项目不一定会按照以上阶段顺序发展，项目可能会一直处于某个阶段，也可能会因为一些特殊原因在多个阶段之间进行变换。

截止到本书编写时，Hyperledger项目下共有12个子项目在运行中，详细信息如表4.1所示。

表4.1 Hyperledger子项目信息表

项 目 名	状 态	依 赖	描 述
Blockchain Explorer	孵化	Fabric, (Sawtooth, Iroha)	区块链Web 浏览器
Cello	孵化	Fabric, (Sawtooth, Iroha)	管理/操作区块链平台
Cello Analytics	孵化	Cello	区块链分析工具
Composer	孵化	Fabric, (Sawtooth, Iroha)	链码编辑器
Fabric	活跃		区块链实现 (Go)
Fabric Chaintool	活跃	Fabric	Fabric的链码工具
Fabric SDK Py	活跃	Fabric	Fabric SDK (Python)
Fabric SDK Go	孵化	Fabric	Fabric SDK (Golang)
Iroha	孵化		区块链实现 (C++)
Sawtooth Lake	孵化		区块链实现 (Python)
Burrow	孵化		模块化EVM合约引擎
Indy	孵化		去中心化身份

项目约定共同遵守的基本原则如下。

- 重视模块化设计，包括交易、合同、一致性、身份、存储等技术场景；
- 代码可读性，保障新功能和模块都可以很容易添加和扩展；
- 发展路线，随着商业化需求的深入和应用场景的丰富，不断增加和演化新的项目。

接下来针对其中几个重要子项目进行进一步的介绍。

1. Fabric

Fabric是一种区块链技术的实现，也是一种基于交易调用和数字事件的分布式共享账本技术。比起其他的区块链技术实现，它采用了模块化的架构设计，支持可插拔组件的开发与使用。其总帐上的数据，由多方参与节点共同维护，并且一旦被记录，账本上的交易信息永远无法被篡改，并支持通过时间戳进行溯源查询。对于其他公有链而言，Fabric引入了成员管理服务，因此每个参与者在进入前均需要提供对应的证书证明身份才能允许访问Fabric系统，同时引入多通道多账本的设计来增强系统的安全性和私密性。与以太坊相比，Fabric采用了强大的Docker容器技术来运行服务，支持比以太坊更便捷、更强大的智能合约服务，以太坊只能通过提供的Solidity语言进行合约编写，而Fabric可以支持多语言的合约便携，例如Go和Java。除此之外，Fabric还提供了多语言的SDK开发接口，让开发者可以自由、便捷地使用其所提供的区块链服务。作为本章的介绍对象，后面还将会深入分析Fabric的架构和运行。

2. Iroha

Iroha是一个受Fabric架构启发而提出的分布式账本项目，该项目在2016年10月13日通过技术指导委员会的批准，进入孵化阶段。它旨在为C++和移动应用开发人员提供Hyperledger项目的开

发环境。该项目希望用C++实现Fabric、Sawtooth Lake和其他潜在区块链项目的可重复使用组件，并且这些组件可以用Go语言进行调用。也就是说，Iroha是对现有项目的一个补充，其长期的目标是实现一个健全的可重用组件库，使Hyperledger技术项目在运行分布式账本时，能自由地选择并使用这些可重复使用的元素。

3. Sawtooth Lake

Sawtooth Lake于2016年4月14日通过TSC批准，是一个由Intel发起的模块化分布式账本平台实验项目，它专为多功能性和可扩展性而设计。Sawtooth Lake提供了一个构建、部署和运行分布式账本的模块化平台，同时支持许可链和非许可链的部署。它包含了一个新的共识算法PoET。PoET与比特币采用的工作量证明算法一样，都是按照一定规则随机选取出一个节点，由该节点来作为区块的记账者，而其他节点则负责验证该区块和执行结果。不同的是，PoET不需要消耗大量的算力和能耗，但是需要CPU硬件支持SGX（Software Guard Extensions）特性。由于PoET算法的硬件限制，因此目前暂时仅适合在生产环境中使用PoET算法。

4. Blockchain Explorer

Blockchain Explorer项目旨在为Hyperledger创建一个用户友好的Web应用程序，用于查询Hyperledger区块链上的信息。包括区块信息、交易相关数据信息、网络信息、链码以及分布式账本中存储的相关信息。项目于2016年8月11日通过TSC批准，之后项目启动进入到孵化阶段。

5. Cello

Cello项目于2017年1月5日通过TSC的批准，进入孵化状态。Cello项目致力于提供一种区块链即服务（Blockchain as a Service, BasS），以此来减少手动操纵（创建和销毁）区块链的工作量。通过Cello，操作者可以使用仪表盘（dashboard）来简单地创建和管理区块链，同时用户（链码开发者）可以通过单个请求立即获取区块链信息。也就是说，为操作者提供了一个简易便捷的区块链操作平台。

4.2 Fabric 简介

Hyperledger Fabric是分布式账本技术（DLT）的独特实现，它可在模块化的区块链架构基础上提供企业级的网络安全性、可扩展性、机密性以及高性能。当前Fabric的最新版本为v1.0，相比先前的v0.6版本，v1.0版本针对安全、保密、部署、维护、实际业务场景需求等方面都进行了很多改进，例如架构设计上的Peer节点的功能分离、多通道的隐私隔离、共识的可插拔实现等，都为Fabric提供了更好的服务支持。因此，本书后面关于Fabric的内容均将基于v1.0版本进行描述。

Hyperledger Fabric v1.0具有以下特性。

- 身份管理（Identity management）：Fabric区块链是一个许可链网络，因此Fabric提供了一个成员服务（Member Service），用于管理用户ID并对网络上所有的参与者进行认证。在

Hyperledger Fabric区块链网络中,成员之间可以通过身份信息互相识别,但是他们并不知道彼此在做什么,这就是Fabric提供的机密性和隐私性。

- ❑ 隐私和保密 (Privacy and confidentiality): Hyperledger Fabric允许竞争的商业组织机构和其他任意对交易信息有隐私和机密需求的团体在相同的许可链网络中共存。其通过通道来限制消息的传播路径,为网络成员提供了交易的隐私性和机密性保护。在通道中的所有数据,包括交易、成员以及通道信息,对于未订阅该通道的网络实体都是不可见且无法访问的。
- ❑ 高效的性能 (Efficient processing): Hyperledger Fabric按照节点类型分配网络角色。为了提供更好的网络并发性,Fabric对事务执行、事务排序、事务提交进行了有效的分离。先于排序之前执行事务可以使得每个Peer节点同时处理多个事务,这种并发执行极大地提高了Peer节点的处理效率,加速了交易到共识服务的交付过程。
- ❑ 函数式合约代码编程 (chaincode functionality): 合约代码是通道中交易调用的编码逻辑,定义了用于更改资产所有权的参数,确保数字资产所有权转让的所有交易都遵守相同的规则和要求。
- ❑ 模块化设计 (Modular design): Hyperledger Fabric实现的模块化架构可以为网络设计者提供功能选择。例如,特定的身份识别、共识和加密算法可以作为可插拔组件插入Fabric网络中,基于此,任何行业或公共领域都可以采用通用的区块链架构。

4.3 核心概念

• 锚节点

锚节点是通道中能被所有对等节点探测并能与之进行通信的一种对等节点。通道中的每个成员都有一个(或多个,以防单点故障)锚节点,允许属于不同成员身份的节点来发现通道中存在的其他节点。

• 区块

区块是一组有序的交易集合,在通道中经过加密(哈希处理)后与前序区块连接。

• 区块链

区块链是一个交易区块经过“哈希连接”结构化的交易日志。对等节点从排序服务收到交易区块,基于背书策略和并发冲突来标注区块的交易为有效或者无效状态,并且将区块追加到对等节点文件系统的哈希链中。

• 合约代码

合约代码是一个运行在账本上的软件,它可以对资产进行编码,其中的交易指令(或者叫业务逻辑)也可以用来修改资产。

• 通道

通道是构建在Fabric网络上的私有区块链，实现了数据的隔离和保密。通道内特定的账本在通道中是与所有对等节点共享的，并且交易方必须通过该通道的正确性验证才能与账本进行交互。通道是由一个“配置块”来定义的。

• 提交

一个通道中的每个对等节点都会验证区块的有序性，然后将区块提交（写或追加）至该通道上账本的各个副本。对等节点也会标记每个区块中的每笔交易的状态是有效或者无效。

• 并发控制版本检查

并发控制版本检查（CCVC）是保持通道中各对等节点间状态同步的一种方法。对等节点并行地执行交易，在交易提交至账本之前，对等节点会检查交易在执行期间读到的数据是否被修改。如果读取的数据在执行和提交之间被改变，就会引发CCVC冲突，该交易就会在账本中被标记为无效，其值不会更新到状态数据库中。

• 配置区块

包含系统链（排序服务）或通道定义成员和策略的配置数据。对某个通道或整个网络的配置修改（比如，成员离开或加入）将导致生成一个新的配置区块并追加到适当的链上。这个配置区块会包含创始区块的内容加上增量。

• 共识

共识是贯穿整个交易流程的广义术语，主要用于确认交易的排序以及交易集本身的正确性。

• 当前状态

总账的当前状态表示其区块链交易日志中所有key的最新值。对等节点会将最近处理过的每笔交易对应修改的value值更新到账本的当前状态，当前状态表示通道中当前最新的key-value值，因此也被称为世界状态。合约代码执行交易提案就是针对的当前状态。

• 动态成员管理

Fabric支持动态添加/移除成员、对等节点和排序服务节点，而不会影响整个网络的操作性。当业务关系调整或因各种原因需添加/移除实体时，动态成员管理至关重要。

• 创世区块

创世区块是初始化区块链网络或通道的配置区块，也是区块链上的第一个区块。

• Gossip协议

Gossip数据传输协议有3项功能：管理对等节点，发现通道上的成员；通道上的所有对等节点间广播账本数据；通道上的所有对等节点间同步账本数据。

- 账本

账本是通道中的区块链和通道中每个节点维护的世界状态的集合。

- 成员管理服务

成员管理服务在许可区块链上认证、授权和管理身份。在对等节点和排序服务节点中运行成员管理服务的代理。

- 排序服务或共识服务

将交易排序放入区块的节点的集合。排序服务独立于对等节点流程之外，并以先到先得的方式为网络上所有的通道做交易排序。排序服务支持可插拔实现，目前默认实现了Solo和Kafka。

- 节点

维护账本并运行合约容器来对账本执行读写操作的网络实体。节点由成员拥有和维护。

4.4 架构详解

Hyperledger是当前业界较为认可的联盟链实现，作为其最重要的子项目，Fabric备受关注。从孵化到发展至今，Fabric的架构设计也在演进过程中逐渐地改进与完善。前面已经对Fabric做了基本内容与功能的介绍，接下来将开始深入探索Fabric，对Fabric最新的总体架构进行分析，并通过与过往架构对比的方式来探讨Fabric新架构的特点和优势。

4.4.1 架构解读

Fabric在架构设计上采用了模块化的设计理念，从图4.1所示的整体逻辑架构来看，Fabric主要由3个服务模块组成，分别是成员服务（Membership Service）、区块链服务（Blockchain Service）和链码服务（Chaincode Service）。其中，成员服务提供会员注册、身份保护、内容保护、交易审计功能，以保证平台访问的安全性和权限管理；区块链服务负责节点的共识管理、账本的分布式计算、账本的存储以及节点间的P2P协议功能的实现，是区块链的核心组成部分，为区块链的主体功能提供底层支撑；链码服务则提供一个智能合约的执行引擎，为Fabric的合约代码（智能合约）程序提供部署运行环境。同时在逻辑架构图中，还能看到事件流（Event Stream）贯穿三大服务组件间，它的功能是为各个组件的异步通信提供技术支持。在Fabric的接口部分，提供了API、SDK和CLI这3种接口，用户可以用来对Fabric进行操作管理。

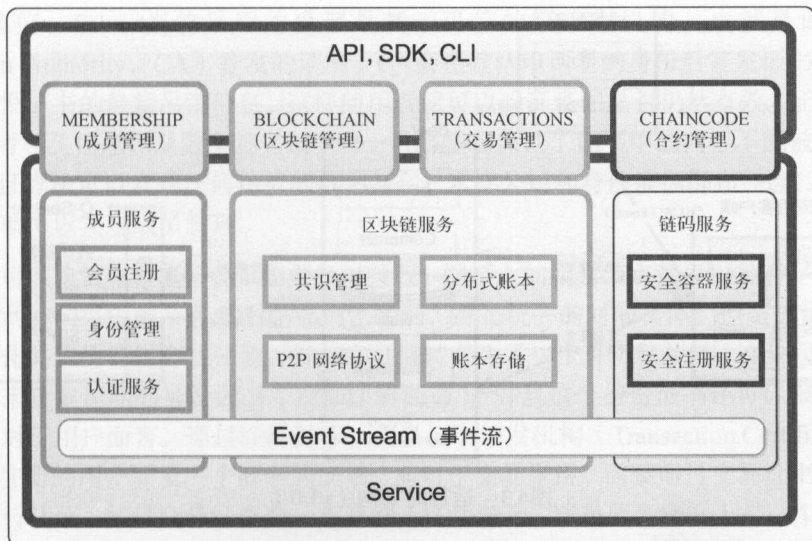


图4.1 逻辑架构图

图4.2和图4.3展示了Fabric运行架构，v0.6版本的结构非常简单，应用-成员管理-Peer呈现三角形关系，系统所有的业务功能均由Peer节点完成。在v0.6版本中，Peer节点承担了太多的业务功能，暴露出了扩展性、可维护性、安全性、业务隔离等方面的诸多问题。因此，在v1.0版本中，官方对架构进行了改进和重构。可以清晰地看到，v1.0版本将共识服务部分从Peer节点中完全分离出来，独立形成一个新的节点提供共识服务和广播服务。同时v1.0版本引入了通道的概念，实现多通道结构和多链网络，带来更为灵活的业务适应性。同时还支持更强的配置功能和策略管理功能，进一步增强系统的灵活性。

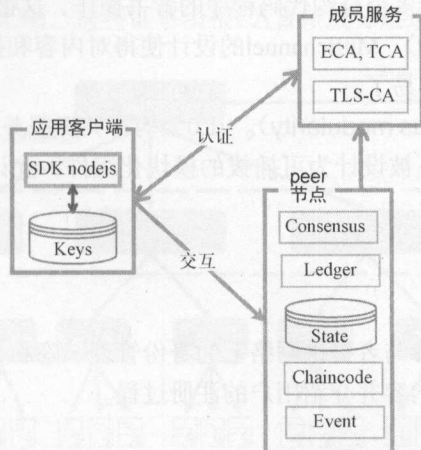


图4.2 运行时架构（v0.6）

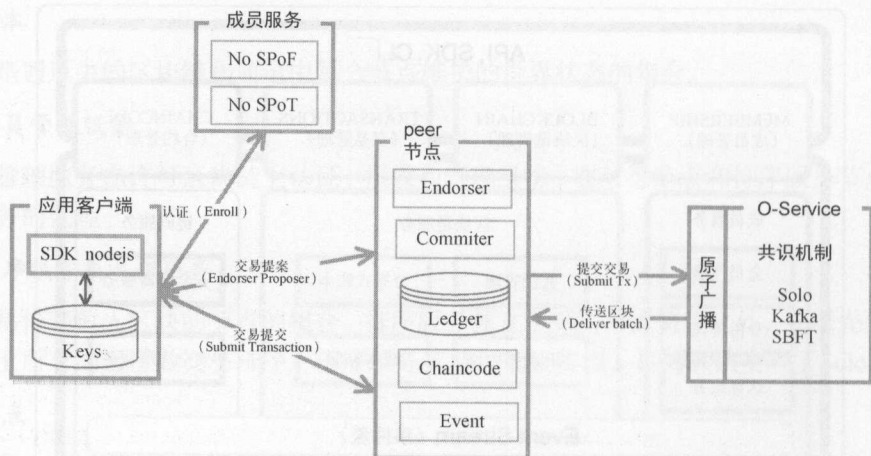


图4.3 运行时架构 (v1.0)

相比v0.6版本，新的架构使得系统在很多方面有了很大的提升，主要有以下的几大优势。

- ❑ 合约代码信任的灵活性（chaincode trust flexibility）。v1.0版本从架构上，将合约代码的信任假设（trust assumptions）与共识服务（ordering service）的信任假设进行了分离。新版本的共识服务可以由一组单独的节点（orderer）来提供，甚至允许出现一些失效节点或恶意节点。而对于合约代码程序而言，它可以指定不同的背书节点，这极大地增强了合约代码的灵活性。
- ❑ 可扩展性（scalability）。在新的架构下，负责为指定合约代码背书的背书节点与共识节点是一种正交的关系，所以相比原来v0.6架构所有业务功能都在Peer节点上执行，v1.0版本架构的扩展性有了很大的提升。尤其是当不同的合约代码所指定的背书节点不存在交集时，系统可以同时进行多个合约代码程序的背书操作，这很好地提高了系统处理的效率。
- ❑ 机密性（confidentiality）。Mutichannel的设计使得对内容和执行状态更新有机密性需求的合约代码的部署变得容易了。
- ❑ 共识模块性（consensus modularity）。v1.0架构将共识服务从Peer节点分离出来独自成为共识节点，共识服务还被设计为可插拔的模块化组件，允许不同共识算法的实现来应用于复杂多样的商业场景。

4.4.2 成员服务

成员服务可以为Fabric的参与者提供网络上的身份管理、隐私、保密性和可审核性的服务。下面重点介绍PKI体系的相关内容并介绍用户的注册过程。

1. PKI体系

PKI（Public Key Infrastructure，公钥基础设施）的目标就是实现不同成员在不见面的情况下

在非许可区块链中，参与者不需要经过授权，网络上的所有节点都可以拥有平等提交交易或者记账的权利，网络中的节点并不存在角色区别，都是统一的对等实体。但是成员服务把PKI体系和去中心化共识协议组合在一起，将非许可链转变为了一个许可区块链。在许可区块链中，实体需要注册来获取长期的身份证书（例如注册证书），并且这个身份证书还可以根据实体类型来进行区分。对于用户而言，通过注册操作，交易证书颁发机构（Transaction Certificate Authority, TCA）会给注册的用户颁发一个匿名的证书，而对于交易来说，需要通过交易证书来对给需要提交的交易进行认证，并且交易证书会一直存储于区块链上。成员服务实际上是一个认证中心，负责为用户提供证书认证和权限管理的功能，对区块链网络中的节点和交易进行管理和认证。

4

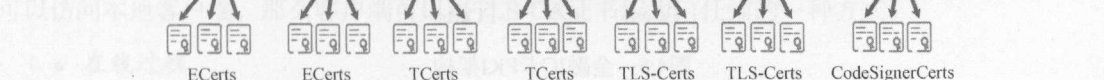


图4.4 成员服务实体组成

下面针对图中的实体进行进一步介绍说明。

Root Certificate Authority (Root CA): 根CA, 代表PKI体系中信任的实体, 同时也是PKI体系结构中的最顶层认证机构。

Enrollment Certificate Authority (ECA): 负责在验证用户提供的注册凭证后发出注册证书 (ECerts)。

Transaction Certificate Authority (TCA): 负责在验证用户提供的注册凭证后发出交易证书 (TCerts)。

TLS Certificate Authority (TLS-CA): 负责颁发允许用户使用其网络的TLS (Transport Layer Security, 传输层安全协议) 证书和凭据。

Enrollment Certificates (ECerts): ECerts是长期证书。它们针对所有角色颁发。

Transaction Certificates (TCerts): TCerts是每个交易的短期证书。它们是由TCA根据授权的用户请求颁发的。此外, TCerts可以被配置为不携带用户身份的信息。它们使得用户不仅可以匿名地参与系统, 还可以防止事务的可链接性。

TLS-Certificates (TLS-Certs): TLS-Certs是用于系统和组件之间进行通信的证书。它们携带其所有者的身份,并用于网络级安全性。

Code Signer Certificates (CodeSignerCerts): 负责对代码进行数字签名, 通过对代码的数字签名来标识软件来源及软件开发者的真实身份, 以此保证代码在签名之后不被恶意篡改。

金融IC卡系统中也使用了PKI体系，它的架构如图4.5所示。与Fabric的PKI体系相比，它没有TCert，每次交易都是使用ECert完成，所以这个系统中的交易是没有匿名的。

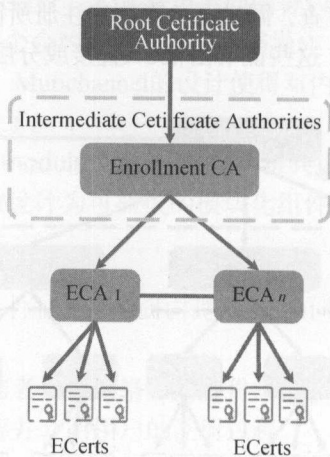
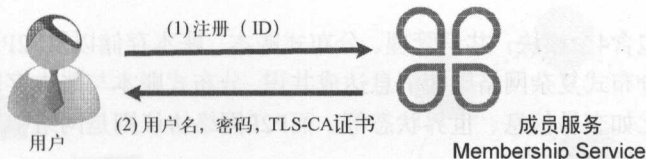


图4.5 金融IC卡PKI架构

2. 用户/客户端注册过程

前面介绍了成员服务的PKI体系的实体及其基本功能，接下来针对具体的用户注册流程做一个简单的介绍。图4.6展示了一个用户登记流程的高层描述，它分为两个阶段：离线过程与在线过程。

离线过程（Offline process）



在线过程（Online process）

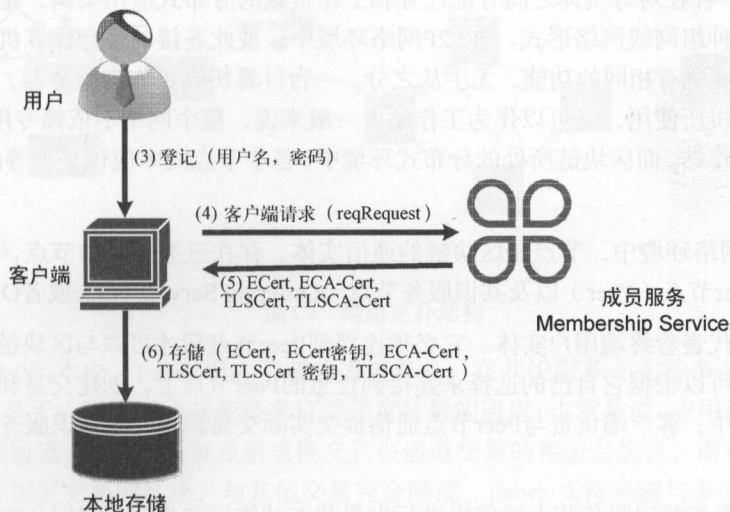


图4.6 用户注册过程

• 离线过程

(1) 每个用户或者Peer节点必须向RA注册机构提供身份证件（ID证明），同时这个流程必须通过带外数据（out-of-band, OOB）进行传输，以提供RA为用户创建（和存储）账户所需的证据。

(2) RA注册机构返回用户有关的用户名和密码，以及信任锚（包含TLS-CA Cert）。如果用户可以访问本地客户端，那么客户端可以将TLS-CA证书作为信任锚的一种方式。

• 在线过程

(3) 用户连接客户端以请求登录系统，在这一过程中，用户将用户名和密码发送给客户端。

(4) 用户端接着代表用户向成员服务发送请求，成员服务接受请求。

(5) 成员服务将包含几个证书的包发送给客户端。

(6) 一旦客户端验证完成所有的加密材料是正确有效的，它就会将证书存储于本地数据库中并通知用户，至此，用户注册完成。

4.4.3 区块链服务

区块链服务包含4个模块：共识管理、分布式账本、账本存储以及P2P网络协议。共识管理用于在多个节点的分布式复杂网络中使消息达成共识，分布式账本与账本存储负责区块链系统中所有的数据存储，比如交易信息、世界状态等。而P2P网络协议则是网络中节点的通信方式，负责Fabric中各节点间的通信与交互。

1. P2P网络

P2P网络是一种在对等实体之间分配任务和工作负载的分布式应用架构，是对等计算模型在应用层形成的一种组网或网络形式。在P2P网络环境中，彼此连接的多台计算机之间都处于对等的地位，各台计算机有相同的功能，无主从之分。一台计算机既可作为服务器，设定共享资源供网络中其他计算机所使用，又可以作为工作站。一般来说，整个网络不依赖专用的集中服务器，也没有专用的工作站。而区块链所处的分布式环境中，各个节点间本应该是平等的，天然适合P2P网络协议。

在Fabric的网络环境中，节点是区块链的通信实体。存在三类不同的节点，分别是客户端节点（Client）、Peer节点（Peer）以及共识服务节点（Ordering Service Node或者Orderer）。

客户端节点代表着终端用户实体。它必须连接到Peer节点后才可以与区块链进行通信交互。同时客户端节点可以根据它自己的选择来连接到任意的Peer节点上，创建交易和调用交易。在实际系统运行环境中，客户端负责与Peer节点通信提交实际交易调用，与共识服务通信请求广播交易的任务。

Peer节点负责与共识服务节点通信来进行世界状态的维护和更新。它们会收到共识服务广播的消息，以区块的形式接收排序好的交易信息，然后更新和维护本地的世界状态与账本。与此同时，Peer节点可以额外地担当背书节点的角色，负责为交易背书。背书节点的特殊功能是针对特定的交易设置的，在它提交前对其进行背书操作。每个合约代码程序都可以指定一个包含多个背书节点集合的背书策略。这个策略将定义一个有效的交易背书（通常情况下是背书节点签名的集合）的充要条件。需要注意的是，存在一个特殊情况，在安装新的合约代码的部署交易中，（部署）背书策略是由一个系统合约代码的背书策略指定的，而不能自己指定。

共识服务节点Orderer是共识服务的组成部分。共识服务可以看作一个提供交付保证的通信组织。共识服务节点的职责就是对交易进行排序，确保最后所有的交易是以同样的序列输出，并提供送达保证服务的广播通信服务。关于共识服务，之后还将详细介绍。

谈完节点的类型，再来看看网络的拓扑结构，在v0.6版本中，整个网络由两类节点构成：VP（Validating Peer）验证节点和NVP非验证节点。如图4.7所示，网络中包含了4个验证节点，并且每个节点还连接着2个非验证节点，整个网络的共识则由4个验证节点构成。在v1.0版本中，网络拓扑结构随着网络节点类型的变化也发生了很大的改变，其中共识服务节点一起组成共识服务，将共识服务抽离出来，而Peer节点中可以分为背书节点或者提交Peer节点，并且它们还可以进行分组，然后整个共识服务与Peer节点所构成的组一起形成新的完成网络。

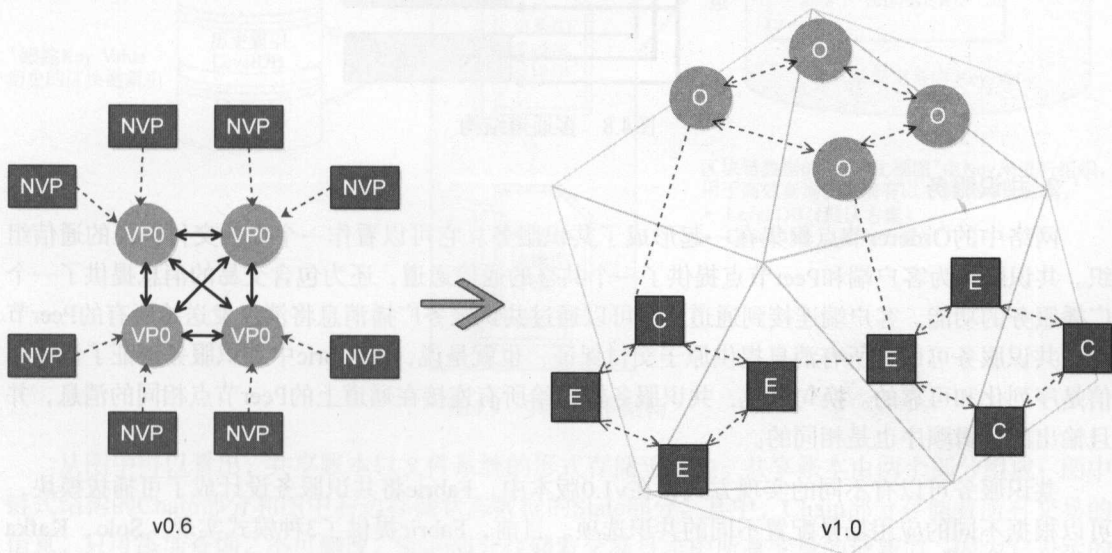


图4.7 网络拓扑结构

同时，在v1.0版本中，Fabric引入了新的通道概念，在共识服务上支持多通道消息传递，使得Peer节点可以基于应用访问控制策略来订阅任意数量的通道；也就是说，应用程序可以指定Peer节点的子集架设通道。这些Peer节点组成提交到该通道交易的相关者集合，而且只有这些Peer节点可以接收包含相关交易的区块，与其他交易完全隔离。Fabric支持多链与多通道，即系统中可以存在多个通道以及多条链，如图4.8所示。应用根据业务逻辑决定将每个交易发送到一个或多个通道，不同通道上的交易不会存在任何联系。

总的来说，Fabric在节点和网络方面的一些重构和新特性使得Fabric的交易处理能力有了很好的增强，而且很好地实现了隐私隔离。

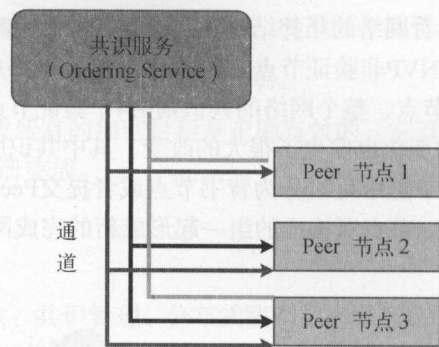


图4.8 多通道结构

2. 共识服务

网络中的Orderer节点聚集在一起形成了共识服务。它可以看作一个提供交付保证的通信组织。共识服务为客户端和Peer节点提供了一个共享的通信通道，还为包含交易的消息提供了一个广播服务的功能。客户端连接到通道后，可以通过共识服务广播消息将消息发送给所有的Peer节点。共识服务可以为所有消息提供原子交付保证，也就是说，在Fabric中共识服务保证了消息通信是序列化 and 可靠的。换句话说，共识服务输出给所有连接在通道上的Peer节点相同的消息，并且输出的逻辑顺序也是相同的。

共识服务可以有不同的实现方式，在v1.0版本中，Fabric将共识服务设计成了可插拔模块，可以根据不同的应用场景配置不同的共识选项。目前，Fabric提供了3种模式实现：Solo、Kafka和BFT。

Solo是一种部署在单个节点上的简单时序服务，主要用于开发测试，它只支持单链和单通道。Kafka是一种支持多通道分区的集群共识服务，可以支持CFT（Crash Faults Tolerance）。它容忍部分节点宕机失效，但是不能容忍恶意节点。其基本实现是基于Zookeeper服务，使用的分布式环境中要求总节点数与失效节点数目满足 $n \geq 2f + 1$ 。BFT则是拜占庭容错模式，这种模式允许在分布式环境中存在恶意节点，也允许节点宕机，但是它要求分布式环境中总节点数据与失效节点数目满足 $n \geq 3f + 1$ 的关系。提供的3种配置模式，从Solo到Kafka再到BFT，面临的分布式的环境越来越复杂，当然这也导致共识服务处理性能有所降低，所以应该根据系统所处环境来选择最优的配置选项。

3. 分布式账本

区块链技术从其底层构造上分析，可以将其定义为一种共享账本技术。账本是区块链的核心组成部分，在区块链的账本中，存储了所有的历史交易和状态改变记录。在Fabric中，每个通道都对应着一个共享账本，而每个连接在共享账本上的Peer节点，都能参与网络和查看账本信息，即它允许网络中的所有节点参与和查看账本信息。账本上的信息是公开共享的，并且在每个peer节点上，都维持着一份账本的副本。图4.9展示了Fabric账本的结构。

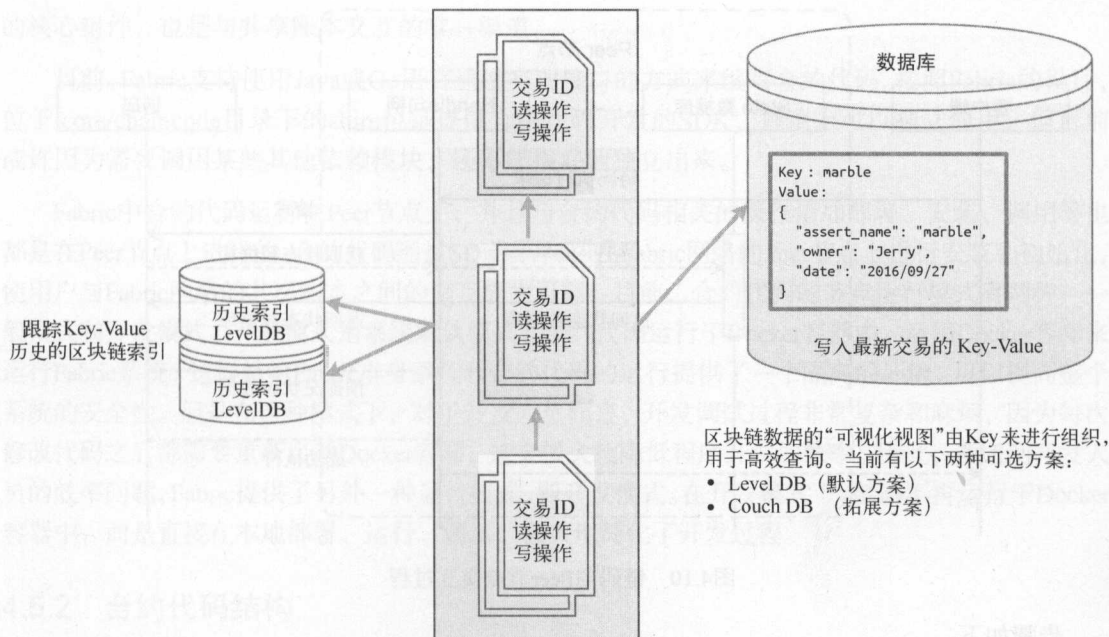


图4.9 共享账本结构

从图中可以看出，共享账本以文件系统的形式存储于本地。共享账本由两个部分组成：图中链式结构的Chain部分和图中右边存储状态数据的State部分。其中，Chain部分存储着所有交易的信息，只可添加查询，不可删改。State部分存储着交易日志中所有变量的最新值，因为它表示的是通道中所有变量键值对的最新值，所以有时称为“世界状态”。

合约代码调用执行交易来更改目前的状态数据，为了使这些合约代码高效交互，设计将最新的键值对数据存储于状态数据库中。默认的状态数据库采用的是Level DB，但是可以通过配置切换到Couch DB或者其他。

4.4.4 合约代码服务

合约代码服务提供了一种安全且轻量级的方式，沙箱验证节点上的合约代码执行，提供安全容器服务以及安全的合约代码注册服务。其运行环境是一个“锁定”和安全的容器，合约代码首先会被编译成一个独立的应用程序，运行于隔离的Docker容器中。在合约代码部署时，将会自动生成一组带有签名的智能合约的Docker基础镜像。在Docker容器中，Peer节点与合约代码交互过程如图4.10所示。

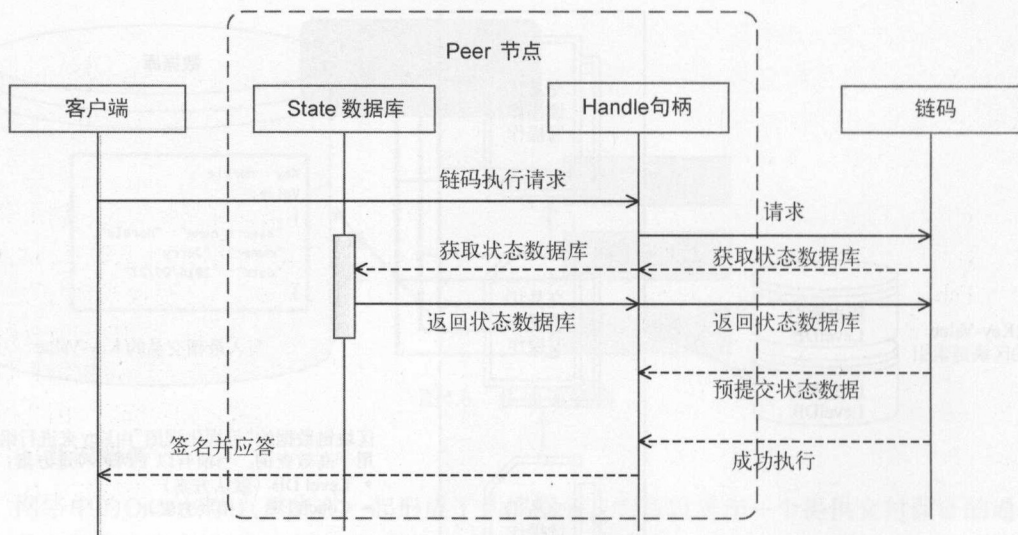


图4.10 链码与Peer节点交互过程

步骤如下。

(1) 合约代码通过gRPC与Peer节点交互，当Peer节点收到请求的输入后，会通过发送一个合约代码消息对象给对应的合约代码。

(2) 合约代码调用Invoke()方法，通过getState()和putState()方法进行读取和写入数据，向Peer节点获取账本状态信息和发送预提交状态。

(3) 合约代码发送最终输出结果给Peer节点，节点对输入和输出信息进行背书签名，完成签名提交。

关于具体的合约代码分析与编写，将在下一节进行详细介绍。

4.5 合约代码分析

通过前述的架构解读部分，可以得知合约代码服务是Fabric架构中的核心组成部分，本节将进一步研究合约代码服务中所运行的合约代码，介绍如何编写、部署及调用具体的合约代码。

4.5.1 合约代码概述

合约代码是区块链上运行的一段代码，是Fabric中智能合约的实现方式。同时在Fabric中，合约代码还是交易生成的唯一来源。共享总账是由区块连接而成的一条不断增长的哈希链，而区块中包含了以Merkle树的数据结构表示的所有的交易信息，可以说交易是区块链上最基础的实体单元。那么交易又是怎样产生的呢？交易只能通过合约代码调用操作而产生，所以合约代码是Fabric

的核心组件，也是与共享账本交互的唯一渠道。

目前，Fabric支持使用Java或Go语言通过实现接口的方式来编写合约代码。按照Fabric的设计，位于/core/chaincode目录下的shim包是提供合约代码开发的SDK，理论上可以独立使用，但目前或许因为需要调用某些其他依赖模块，还不能很好地独立出来。

Fabric中合约代码运行在Peer节点上，并且与合约代码相关的操作诸如部署、安装、调用等也都是在Peer节点上进行的。合约代码通过SDK或者CLI在Fabric网络的Peer节点上进行安装和初始化，使用户与Fabric网络的共享账本之间的交互成为可能。目前，合约代码的节点运行模式有两种：一般模式和开发模式。一般模式是系统默认模式，合约代码运行于Docker容器中。运用Docker容器来运行Fabric系统，这样就给Fabric本身系统和合约代码的运行提供了一个隔离的环境，可以提高整个系统的安全性。但是在这种模式下，对于开发人员而言，开发调试过程非常复杂和麻烦，因为每次修改代码之后都需要重新启动Docker容器，这会极大地降低程序开发的效率。因此，考虑到开发人员的效率问题，Fabric提供了另外一种运行模式，即开发模式。在开发模式下，链码不再运行于Docker容器中，而是直接在本地部署、运行、调试，极大地简化了开发过程。

4

4.5.2 合约代码结构

合约代码是Fabric开发中最主要的部分之一，通过合约代码可以实现对账本和交易等实体的交互与操作，同时实现各种业务逻辑。目前，合约代码支持Go语言和Java语言进行编写，通过实现链码接口的方式来编写合约代码程序。下面以Go语言为例。

合约代码的结构主要包括以下3个方面。

1. 链码接口

在Fabric v1.0版本中，链码接口包含两个方法：Init()方法和Invoke()方法。Init()方法会在第一次部署合约代码时进行调用，有点类似于类中的构造方法。就如同其方法名所表达的，Init()方法中一般执行一些合约代码需要的初始化操作。Invoke()方法则是在调用合约代码方法进行一些实际操作时调用，每次调用会被视为一次交易执行。完整的详细交易流程将在4.6节进行介绍。Go语言中的链码接口代码如下所示：

```
Type Chaincode interface {  
    // 初始化工作，一般情况下仅被调用一次  
    Init(stub ChaincodeStubInterface) pb.Response  
    // 查询或更新world state，可被多次调用  
    Invoke(stub ChaincodeStubInterface) pb.Response  
}
```

2. API方法

当合约的Init或者Invoke接口被调用时，Fabric传递给合约shim.ChaincodeStubInterface参数并返回pb.Response结果，这些参数可以通过调用API方法去操作账本服务，产生交易信息或者调用其他的合约代码。

目前API方法定义在/core/chaincode目录下的shim包中，并且可以由以下命令生成：

```
godoc github.com/hyperledger/fabric/core/chaincode/shim
```

其中主要的API方法可以分为5类，分别是State读写操作、Args读写操作、Transaction读写操作、链码相互调用以及Event设置。下面通过表4.2来显示这些方法及其所对应的功能。

表4.2 API方法

API方法	功 能
GetState((key string)	获取最新state
PutState(key string, value []byte)	新增state
DelState(key string)	删除state
GetStateByRange(startKey, endKey string)	获取state
GetStateByParGalCompositeKey (objectType string, keys []string)	获取state
GetQueryResult (query string)	查询结果
GetHistoryForKey(key string)	查询历史
CreateCompositeKey (objectType string, attributes []string)	创建复合键
SplitCompositeKey(compositeKey string)	切割复合键
GetArgs()	读取参数
GetStringArgs()	读取参数字符串
GetFuncGonAndParameters()	读取函数和参数
GetArgsSlice()	读取参数切片
GetCreator()	读取创建者
GetTransient()	读取交易信息
GetBinding()	读取Bind
GetTxTimestamp()	读取时间戳
GetTxID()	读取TxId
InvokeChaincode (chaincodeName string, args [][]byte, channel string)	调用链码
SetEvent(name string, payload []byte)	设置事件流
NewLogger(name string) *ChaincodeLogger	新建日志
Start(cc Chaincode)	启动链码
SetLoggingLevel(level LoggingLevel)	设置日志级别

3. 链码返回信息

链码是以protobuffer的形式返回的，定义如下所示：

```
message Response {
// 状态码
    int32 status = 1;
    // 响应码信息
    string message = 2;
    // 响应内容
    bytes payload = 3;
}
```

链码还会返回事件信息，包括 Message events和Chaincode events，定义如下所示：

```
messageEvent {
    oneof Event {
        Register register = 1;
        Block block = 2;
        ChaincodeEvent chaincodeEvent = 3;
        Rejection rejection = 4;
        Unregister unregister = 5;
    }
}
messageChaincodeEvent {
    string chaincodeID = 1;
    string txID = 2;
    string eventName = 3;
    bytes payload = 4;
}
```

一旦完成了链码的开发，有两种方式可以与链码交互：通过SDK或者通过CLI命令行。通过CLI命令行的方式交互将在下一节进行介绍，SDK的交互可以参考第5章的案例。

4.5.3 CLI 命令行调用

编写完合约代码之后，就要了解如何部署合约代码以及如何调用合约代码。要想进行部署合约代码等相关操作，必然需要启动Fabric系统。Fabric提供了CLI接口，支持以命令行的形式完成与Peer节点相关的操作。通过CLI接口，Fabric支持Peer节点的启动停止操作、合约代码的各种相关操作以及通道的相关操作。

当前Fabric所支持的CLI命令如表4.3所示。

表4.3 CLI命令

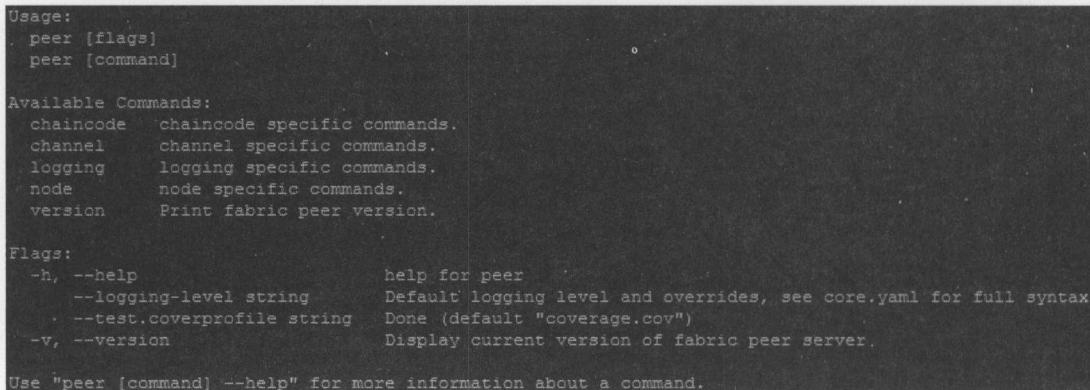
命令行参数	功 能	命令行参数	功 能
Version	查看版本信息	chaincode invoke	调用链码
node start	启动节点	chaincode query	查询链码
node status	查看节点状态	channel create	创建通道
node stop	停止节点	channel join	加入通道
chaincode deploy	部署链码		

同时通过以下命令，查看更多与Peer命令相关的信息。

```
# 此命令需要
cd /opt/gopath/src/github.com/hyperledger/fabric
build /bin/peer
```

```
# 或者进入启动网络后进入cli容器
docker exec -it cli bash
# 进入cli容器后运行peer命令
peer
```

在运行以上命令之后，将看到如图4.11所示的信息。



```
Usage:
  peer [flags]
  peer [command]

Available Commands:
  chaincode  chaincode specific commands.
  channel    channel specific commands.
  logging    logging specific commands.
  node       node specific commands.
  version    Print fabric peer version.

Flags:
  -h, --help                help for peer
  --logging-level string    Default logging level and overrides, see core.yaml for full syntax
  --test.coverprofile string Done (default "coverage.cov")
  -v, --version             Display current version of fabric peer server.

Use "peer [command] --help" for more information about a command.
```

图4.11 Peer命令行参数

可以按照图中所示，运行`peer [command] -help`命令查看更详细的命令介绍信息。

4.5.4 链码执行泳道图

链码执行过程如图4.12所示。

- ❑ 客户端（SDK/CLI）创建交易提案，包含链码函数和调用参数，并以proto消息格式发送到背书节点。
- ❑ 背书节点调用SHIM包的方法创建链码仿真交易执行内容。
- ❑ 背书节点初始化合约、调用参数，基于读取和写入的Key生成读写操作集。
- ❑ 背书集群节点模拟提案执行：执行读操作，向账本发送查询状态数据库的请求；模拟写操作，获取Key的value值版本号，模拟更新状态数据。
- ❑ 若返回执行成功，则执行背书操作；若返回失败，则推送错误码500。
- ❑ 背书节点对交易结果执行签名，将提案结果返回给客户端（SDK/CLI），提案结果包括执行返回值、交易结果、背书节点的签名和背书结果（同意或拒绝）。

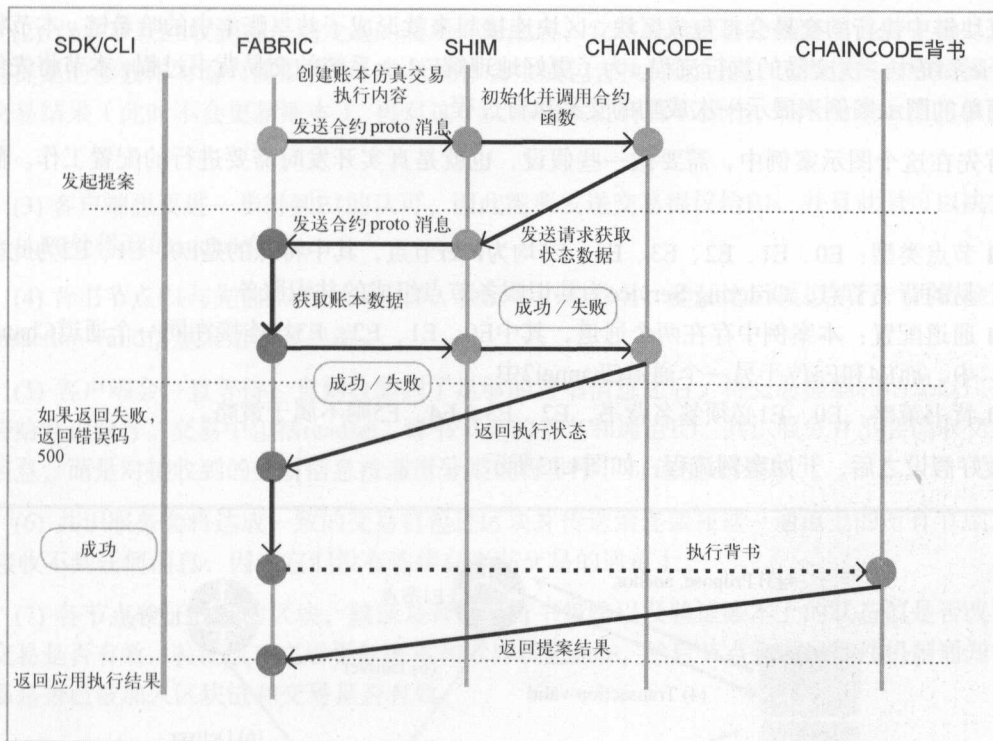


图4.12 链码执行流程图

关于合约代码编写的内容，可以查看项目/examples/chaincode/下的示例合约代码了解更多。

4.6 交易流程

本节主要分析Fabric中的交易背书过程，首先介绍了Fabric交易背书过程的机制，然后通过一个简单的案例描述了其通用流程，之后详细分析背书过程，最后简单地介绍了Fabric的背书策略。

4.6.1 通用流程

在Fabric系统中，交易就是一次合约代码的调用，交易可能有如下两种类型。

- 部署交易。部署交易创建新的合约代码，并且用一个程序作为参数。当一个部署交易成功执行时，合约代码被安装到区块链上。
- 调用交易。调用交易在先前部署的交易上下文中执行操作。调用交易指的是合约代码和它提供的一个或多个功能。当成功地执行调用交易时，合约代码执行指定的函数，这些函数执行时可能修改相应的状态，并返回输出。

区块链中执行的交易会打包成区块，区块连接起来就形成了共享账本中的哈希链。本节将介绍Fabric系统中一次交易的执行流程。为了更好地理解Fabric系统的交易背书过程，本节将先使用一个简单的图示案例来展示一次成功的交易执行过程。

首先在这个图示案例中，需要做一些假设，也就是真实开发时需要进行的配置工作。假设如下。

- ❑ 节点类型：E0、E1、E2、E3、E4、E5均为Peer节点，其中特殊的是E0、E1、E2为此次交易的背书节点，Ordering Service为共识服务节点组成的共识服务。
- ❑ 通道配置：本案例中存在两个通道，其中E0、E1、E2、E3均连接在同一个通道Channel1中，而E4和E5位于另一个通道Channel2中。
- ❑ 背书策略：E0、E1必须签名背书，E2、E3、E4、E5则不属于策略。

做好假设之后，开始案例流程，如图4.13所示。

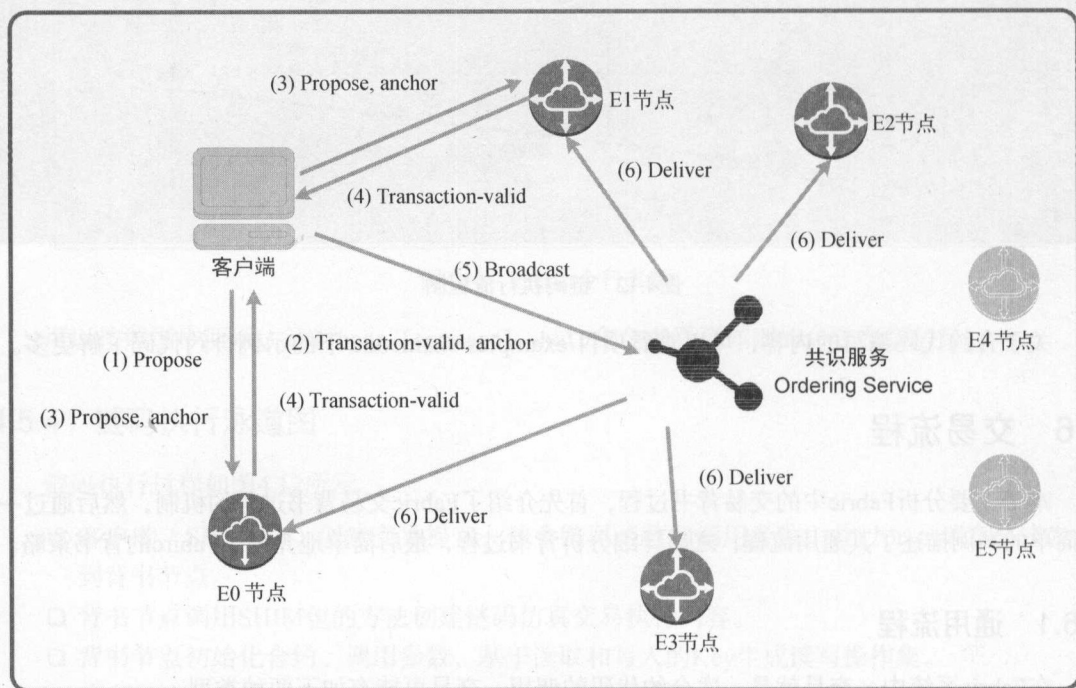


图4.13 交易背书通用流程

(1) 客户端应用通过SDK发送出一个交易提案（Transaction Propose）给背书节点E0。它用来接收智能合约中相关功能函数的请求，然后更新账本数据（即资产的键/值）。同时在发送前客户端会将这一交易提案打包为一种可识别的格式（如gRPC上的protocol buffer），并使用用户的加密凭证为该交易提案签名。

(2) 背书节点E0收到客户端发送的交易提案之后，它将先验证客户端签名是否正确，然后将交易提案的参数作为输入模拟执行，执行操作会生成包含执行返回值、读操作集合和写操作集合的交易结果（此时不会更新账本），再对这个交易提案进行背书操作，附上anchor信息发送回客户端。

(3) 客户端想要进一步得到E1的认可，因此需要发送交易提议给E1，并且此时可以决定是否附上从E0处得到的anchor信息。

(4) 背书节点E1与先前E0的方式一样，验证客户端签名，验证之后模拟执行，再将验证后的Transaction-valid信息发送回客户端。

(5) 客户端会一直等待，直到收集到了足够的背书信息之后，将交易提案和结果以广播的形式传给共识服务。交易中包括readset、背书节点的签名和通道ID。共识服务并不会读取交易的详细信息，而是对接收到的交易信息按通道分类进行排序，打包生成区块。

(6) 共识服务会将达成一致的打包进区块并传送给连接在这一通道上的所有节点，E4和E5接收不到任何消息，因为它们没有连接在当前交易的通道上。

(7) 各节点验证收到的区块，验证是否满足背书策略以及验证账本上的状态值是否改变来判断交易是否有效。验证成功之后更新账本和世界状态state，然后节点会通过事件机制通知客户端交易是否已被加入区块链和交易是否有效。

4.6.2 流程详解

在Fabric中，交易是就指是一次链码调用，下面将详细分析一次交易背书的过程。

1. 客户端发送交易提议给指定背书节点

为了调用一个交易，客户端会向它所选择的一组背书节点发送一个PROPOSE消息（这些消息可能不是同时发送的，比如上一节的例子）。对于如何选择背书节点集合，client可以通过Peer使用给定chaincodeID的背书节点集合，反过来也可以通过背书策略获取背书节点集合。例如，这个交易会被客户端通过chaincodeID发送给所有相关的背书节点。即便如此，某些背书节点可能离线，其他的可能反对，所以选择不签署该交易。提交客户端会通过有效的背书节点来尽力满足背书策略。

下面，本节将先详细说明PROPOSE消息的格式，然后讨论提交客户端和背书节点间可能的交互模式。

(1) PROPOSE消息格式

一条PROPOSE的格式为PROPOSE = <PROPOSE, tx, [anchor]>，包含着两个参数，tx交易消息字段是必需的，而anchor是可选的参数。下面对这两个参数进行详细分析。

tx参数包含了与交易相关的各种信息，字段格式如下：

`tx=<clientID, chaincodeID, txPayload, timestamp, clientSig>`

- `clientID`: 客户端ID
- `chaincodeID`: 调用操作合约ID
- `txPayload`: 包含交易信息的载体
- `timestamp`: 时间戳
- `clientSig`: 客户端签名

而对于`txPayload`字段, 调用交易和部署交易的详细信息会有些不同。

如果当前交易是部署交易, `txPayload`只包含2个字段:

`txPayload = <operation, metadata>`

- `operation`: 指合约代码调用的函数和参数
- `metadata`: 指与此次调用相关的其他属性

如果当前交易是调用交易, `txPayload`还将包含一个`policies`字段:

`txPayload = <operation, metadata, policies >`

- `operation`: 指合约代码调用的函数和参数
- `metadata`: 指与此次调用相关的其他属性
- `policies`: 指与合约代码相关的策略, 比如背书策略

`anchor`参数中包含了`readset` (一个从原始账本中读取到的版本依赖键值对集合), 也就是世界状态中的版本化依赖。如果客户端发送的`PROPOSE`消息中携带了`anchor`参数, 那么背书节点还需要验证`anchor`参数中是否与本地匹配。

同时`tx`字段的加密哈希`tid`还会被所有节点用来作为交易的标识 (`tid=HASH(tx)`), 并且客户端会将它存储在内存中一直等待背书节点响应。

(2) 消息模式

因为客户端的消息是需要发送给一组背书节点的, 所以对于它的发送顺序是可以由客户端控制的。例如, 通常情况下, 客户端会先发送给单个背书节点不携带`anchor`参数的`PROPOSE`消息, 背书节点接收后, 会处理消息并加上`anchor`参数返回给客户端。然后客户端将携带`anchor`参数的`PROPOSE`消息发送给剩下的其他背书节点。而另一种模式, 客户端会直接将不携带`anchor`参数的`PROPOSE`消息发送给背书节点集合, 等待它们的返回。客户端可以自由选择消息模式来进行与背书节点间的交互。

2. 背书节点模拟交易执行, 然后产生一个背书签名

背书节点收到客户端的`<PROPOSE, tx, [anchor]>`消息之后, 它会先验证客户端的签名, 验证通过后会模拟执行交易的内容。需要注意的是, 如果客户端指定了`anchor`字段, 那么需要验证本地KVS中相应键的值, 只有与`anchor`参数中的一致时, 背书节点才会模拟执行交易。

模拟执行将会通过调用chaincodeId对应的合约代码来试验性的执行txPayload中的操作，同时还会获取背书节点本地维护的世界状态的一个副本。在执行完成后，背书节点会更新readset和writeset（存储着状态更新）两个键值对的集合的信息，这个机制在DB数据库中也被称为MVCC+postimage info。具体键值对操作如下。

给定背书节点执行交易之前的状态s，对于交易读取的每个键k，(k,s(k).version)将会被添加到 readset 中。

对于交易修改的每个键 k，(k,v') 将会被添加到 writeset 中，其中 v'是更新后的新值。另外，v'也可以是相对于之前值(s(k).value)的差值。

在模拟执行之后，Peer节点会根据所谓的背书逻辑来决定是否为这一交易进行背书，默认情况下，Peer节点会接收tran-proposal消息并简单地为其签名。然而背书逻辑可以被设置，例如，Peer节点会将tx作为输入，与遗留系统进行交互，来决定是否为这一交易背书。

如果背书逻辑决定背书这个交易，它就会发送<TRANSACTION-ENDORSED, tid, tran-proposal, epSig>消息给客户端：

4

```
tran-proposal := (epID,tid,chaincodeID,txContentBlob,readset,writeset)
```

□ txContentBlob: 交易信息txPayload

□ epSig: 背书节点的签名

如果背书逻辑决定拒绝为这个交易背书，它则会发送(TRANSACTION-INVALID, tid, REJECTED)消息给客户端。

需要注意的是，背书节点模拟执行不会更改任何的哈希链与世界状态信息，它只是模拟执行，然后将操作所引起的状态改变存储于writeset中。

3. 客户端收集交易背书后并通过共识服务广播

在一定的时间间隔内，如果客户端收到了“足够的”背书节点发回的背书消息(TRANSACTION-ENDORSED, tid, *, *)，则背书策略被满足，那么这笔交易就会被认为背书成功，需要注意的是此时还没有提交。

否则，如果一定时间间隔内没有收到足够多的背书消息，那么客户端就会抛弃该笔交易或者稍后进行重试。

对于有效的背书成功的交易，客户端会通过broadcast(blob)方法调用共识服务，其中blob指的就是背书消息。如果client没有直接调用共识服务的能力，它可以选择某个Peer节点代理调用，当然这个Peer节点必须是可信的，否则这个交易可能会被视为背书无效。

4. 共识服务传送区块给Peer节点

在共识服务对交易进行排序并达成区块之后，共识服务将会触发deliver(seqno, prevhash, blob)事件，然后将这一区块广播给所有链接在Fabric和同一通道上的Peer节点。

Peer节点在收到共识服务广播的区块之后会进行两类校验。

第一类是通过(blob.tran-proposal.chaincodeID)指向的合约代码所包含的背书策略来验证blob.endorsement是否有效；第二类则是在完成第一类验证之后，还将验证blob.endorsement.tran-proposal.readset集合是否正确。

针对readset集合的验证，根据“一致性”和“隔离保证”，可以采用不一样的方式。如果在合约代码中未指定相应的背书策略，那么可串行化（Serializability）则是默认的验证方式。对于可串行化要求每个readset中的键的版本要与state中的版本对应，然后拒绝不满足条件的交易。

假如上面的验证都通过了，这个交易就可以被视为有效或者已提交的了。在验证之后，Peer节点就会在peerLedger账本对应的位掩码中用1标记这笔交易，并将writeset中的更新应用到Fabric区块链的state世界状态中。而如果验证失败，那么这个交易就被认为是无效的，Peer节点则会在peerLedger的位掩码中用0标记这笔交易，并且无效的交易不会引起任何改变更新。

在共识服务的保证下，上述流程会保证所有正常的Peer节点在执行一个deliver事件之后拥有相同的世界状态。换句话说，所有正确的节点将会收到一个完全一样的deliver事件的序列。至此，本次交易流程结束。

4.6.3 背书策略

Fabric所提供的背书策略机制是用于指定区块链节点交易验证的规则。每当背书节点收到交易请求的时候，系统就会通过VSCC（Validation System Chaincode，系统合约代码验证）对交易的有效性进行验证。在交易流程中，一个交易可能会包含来自于背书节点的一个或多个背书，而VSCC机制将会根据以下规则决定交易的有效性。

- ❑ 背书的数量是否符合要求；
- ❑ 背书是否来自预期的来源；
- ❑ 所有来自背书节点的背书是否有效（即它们是否来自预期消息上的有效证书的有效签名）。

背书策略就是用来指定以上的背书数量要求和背书来源预期集合。每个背书策略由两个部分组成，原则（principal）和定限闸（threshold gate）。原则P用来识别预期签名的实体；定限闸T有两个输入参数， t 表示背书数量， n 表示背书节点列表，即满足 t 的条件，背书节点属于 n 。例如，T(2, 'A', 'B', 'C')表示需要获得2个以上来自于'A', 'B', 'C'的背书。T(1, 'A', T(2, 'B', 'C'))表示需要收到来自'A'的背书或者来自'B'和'C'的两个背书。

在CLI命令行交互中，背书策略的表示语法是EXPR([E, E...])，EXPR有两个选项AND或者OR，其中AND表示“与”，表示每个都需要，而OR则表示“或”。比如AND('Org1.member', 'Org2.member', 'Org3.member')表示请求3个组的签名，OR('Org1.member', 'Org2.member')表示请求两个组中任意一个的签名即可。而OR('Org1.member', AND('Org2.member', 'Org3.member'))表示有两种选择，第一

种是请求组织1的签名，第二种是请求组织2和组织3的签名。

在使用CLI与区块链交互时，在命令后使用-P选项即可为执行的合约代码指定相应的背书策略，例如下面的合约代码部署命令：

```
peer chaincode deploy -C testchainid -n mycc -p $ORDER_CA -c  
'{"Args":["init","a","100","b","200"]}' -P "AND('Org1.member', 'Org2.member')"
```

表示部署合约代码mycc需要请求组织1和组织2的签名。

Fabric未来对于背书策略会进一步增强改进，除了目前通过与MSP的关系来识别原则，Fabric计划添加OU（Organization Unit）的形式来完成当前证书的功能，同时计划对背书策略的语法进行改进，使用更直观的语法。

4.7 本章小结

本章对Hyperledger Fabric进行了深入解读，有助于读者深入理解Fabric的底层实现原理。首先，介绍了Hyperledger及其子项目的发展现状及管理模式，重点介绍了Hyperledger Fabric。之后，对Hyperledger Fabric架构进行了深入分析，从成员服务、区块链服务以及合约代码服务三个方面探讨Hyperledger Fabric的架构组成与特点，给出了Fabric架构设计和模块组件。然后，给出了链码的代码结构、调用方式和执行流程。最后，对交易背书流程展开了详细分析。

Hyperledger Fabric应用开发基础

5

第4章对Hyperledger Fabric进行了深入解读，读者应对Fabric的核心原理有了基本的认识。在此基础上，本章将主要讲解基于Fabric进行区块链应用开发的最佳实践，从应用开发的角度给出Fabric环境部署、链码开发指南以及CLI和SDK应用开发实例，通过理论与实践相结合的方式，使读者能够更好地基于Fabric进行应用开发。

5.1 环境部署

环境部署是开发实战的第一步，只有成功搭建好开发环境，才能继续后面的实践内容。本节将给出搭建Fabric开发环境的详细过程，为后面的开发实战提供支持。

5.1.1 软件下载与安装

需要安装的必备软件主要有Oracle VM VirtualBox、Vagrant和Git。下面我们分别进行介绍。

1. Oracle VM VirtualBox

Oracle VM VirtualBox是Oracle公司推出的一款虚拟机软件，可以让用户在日常操作系统下，利用虚拟机来安装其他的操作系统。它是一个开源免费的虚拟机软件，可以很好地适应各种跨平台开发的需求，对于开发人员而言是一个很好的工具。而且它允许在一个计算机上同时运行多个虚拟操作系统（如Windows、Linux、Solaris等），用户在使用时只需在不同的窗口间进行切换，就可以轻松在不同的系统上进行开发操作，这是在新的物理机上安装系统所不能带给我们的。

要使用Oracle VM VirtualBox，用户首先要去官方网站（<https://www.virtualbox.org/wiki/Downloads>）下载安装包。VirtualBox支持多平台，提供了各类操作系统的下载安装方案，这里以当前用户群最多的Windows 10 64bit操作系统环境为例，选择 Windows hosts版本进行下载安装，如图5.1所示。

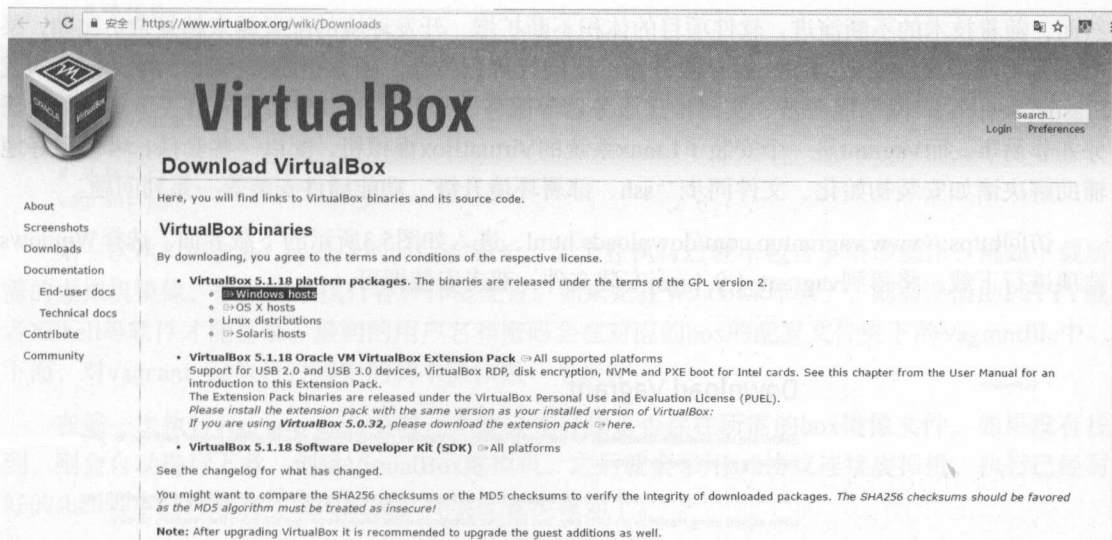


图5.1 选择合适的VirtualBox版本下载

在下载完成之后会得到一个名称为VirtualBox-5.1.18-114002-Win.exe的安装包文件，双击进行安装，按照提示信息选择安装配置，如果没有特殊需求，选择默认即可。安装成功之后进入软件主界面，如图5.2所示。

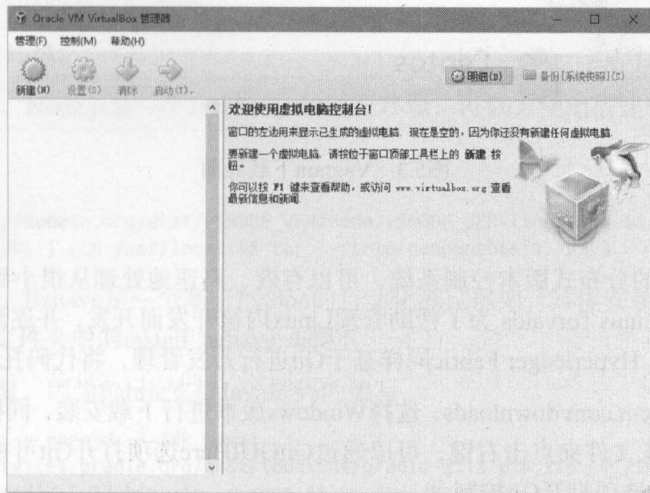


图5.2 VirtualBox主界面

2. Vagrant

Vagrant是一个可创建轻量级、高复用性和便于移植的开发环境的工具，用于创建和部署虚拟化开发环境。在当前的真实开发中，要对某个项目进行开发，第一步往往都是先配置好开发环境。

然而,随着技术的不断演进,软件项目的体积不断扩增,开发环境的配置越来越复杂。比如,某个项目可能需要涉及数据库、缓存服务器、反向代理服务器、搜索引擎服务器(例如Sunspot或Elastic Search)、网站服务器、实时推送服务器等服务。因此,很多时候完成第一步环境配置任务并非易事。而Vagrant是一个安装了Linux系统的VirtualBox虚拟机,配以一些套件,可以很好地辅助解决诸如安装初始化、文件同步、ssh、部署环境升级、功能插件安装等一系列问题。

访问<https://www.vagrantup.com/downloads.html>,进入如图5.3所示的下载界面。选择Windows选项进行下载,将得到vagrant_1.9.4.msi安装文件,双击安装即可。



图5.3 Vagrant下载界面

3. Git

Git是一个开源的分布式版本控制系统,可以有效、高速地处理从很小到非常大的项目版本管理,Linux创始人Linus Torvalds 为了帮助管理Linux内核开发而开发,并逐渐发展成全球最受欢迎的版本控制软件。Hyperledger Fabric同样基于Git进行开发管理,将代码托管于GitHub上。

前往<https://git-scm.com/downloads>,选择Windows版本进行下载安装,同样默认操作即可。安装完成之后,在任意文件夹点击右键,可以通过GitGUIHere选项打开Git可视化操作界面,也可以通过Git Bash Here选项打开Git控制台。

5.1.2 开发环境搭建

在必备软件安装完成后,可以开始使用Vagrant创建开发环境的虚拟机。首先选择一个文件夹,使用以下命令将Hyperledger Fabric项目克隆到本地并启动Vagrant。

```
# 克隆项目
git clone https://github.com/hyperledger/fabric.git
#进入vagrant虚拟机目录
cd fabric/devenv
# 启动vagrant
vagrant up
# 连接virtualbox虚拟机
vagrant ssh
```

第一次执行vagrant up命令的时间会较长，因为在执行过程中包含了很多操作，例如下载所需的虚拟机镜像，根据脚本执行各种环境配置。如果是在Windows环境下，则需要借助PuTTY或者XShell等软件才能登录。最初的用户名和密码会在对应的box的配置文件夹下的Vagrantfile中。下面，对vagrant up执行过程中的脚本操作做一个探究。

在第一次执行vagrant up命令时，系统会先寻找是否存在所需的box镜像文件，如果没有找到，则会自动进行下载，创建VirtualBox虚拟机，之后就会采用ssh协议连接虚拟机，执行已经写好的shell脚本进行开发环境的配置。环境配置步骤如下。

(1) 更新系统。

```
apt-get update
```

(2) 安装一些基础的工具软件。

```
apt-get install -y build-essential git make curl unzip g++ libtool
```

(3) 安装Docker和docker-compose。

(4) 安装Go语言环境。

(5) 安装Node.js，Node.js是一个JavaScript的运行环境，可以方便地搭建响应速度快、易于扩展的网络应用。

```
NODE_VER=6.9.5
NODE_URL=https://nodejs.org/dist/v$NODE_VER/node-v$NODE_VER-linux-x64.tar.gz
curl -sL $NODE_URL | (cd /usr/local && tar --strip-components 1 -xz )
```

(6) 安装Behave，Behave是一个基于Python的自动化测试框架，具体安装可以参考Fabric项目根目录fabric/script/文件夹中的install_behave.sh脚本。

(7) 安装Java环境，因为Fabric支持Java链码的编写。

```
apt-get install -y openjdk-8-jdk maven
wget https://services.gradle.org/distributions/gradle-2.12-bin.zip -P /tmp --quiet
unzip -q /tmp/gradle-2.12-bin.zip -d /opt && rm /tmp/gradle-2.12-bin.zip
ln -s /opt/gradle-2.12/bin/gradle /usr/bin
```

(8) 进行一些杂项配置，开发环境配置完成。

Vagrant提供了一种便捷且全面的开发环境搭建方式，它需要基于VirtualBox虚拟机，然而最终应用往往是运行在物理服务器上的。而针对物理机上Fabric开发环境的构建，Fabric最核心的两

项是Go语言环境和Docker环境的构建。Fabric源码采用Go语言进行编写，而Fabric应用则均以Docker容器的方式运行。至于其他的工具与软件，可以根据开发时的需求自行选择。

5.1.3 Go 和 Docker

本节将针对Go语言环境和Docker环境的搭建进行详细的介绍。

1. Go语言环境

Go语言是谷歌2009年发布的第二款开源编程语言，专门对多处理器系统应用程序的编程进行了优化，使用Go语言编译的程序具有媲美C或者C++代码的速度，同时还更加安全，支持并发操作。Go语言是一门被谷歌寄予厚望的优秀编程语言，其设计是让软件充分地发挥多核心处理器同步多工的优势，并解决面向对象程序设计的麻烦。它拥有着极致精简的语法，具有现代编程语言的特色，如垃圾回收机制可帮助开发者处理琐碎麻烦的内存管理问题。现在的云平台大部分都是基于Go语言进行开发的，比如现在的Docker容器。区块链技术的实现也都以Go语言作为主流开发语言，比如以太坊的Go语言客户端geth，还有我们正在介绍的Hyperledger Fabric，也是使用Go语言进行开发的。下面我们介绍Go语言环境的配置。

(1) 首先去官网下载Go语言安装包，访问<https://golang.org/dl/>，如图5.4所示。

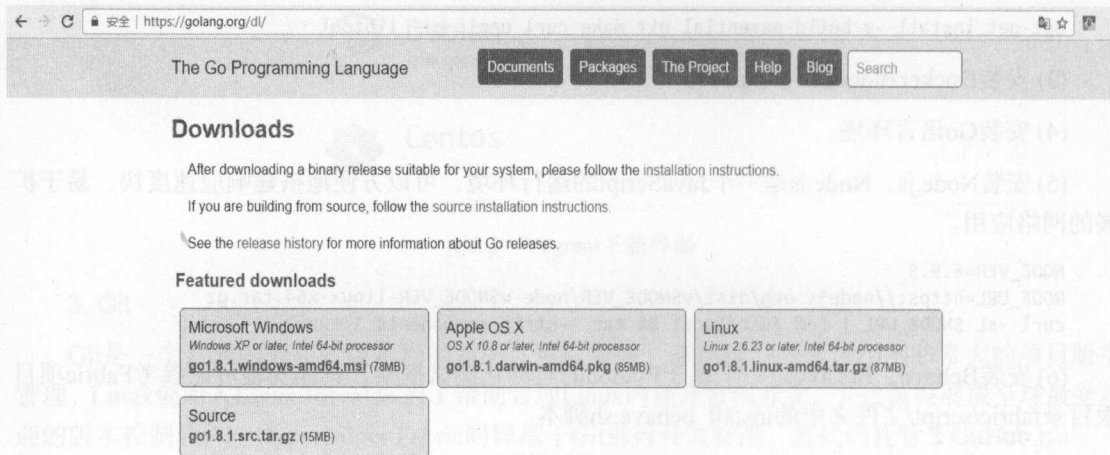


图5.4 Go语言下载界面

(2) 选择Linux版本，复制下载链接，使用以下命令下载。

```
mkdir Download
cd Download
wget https://storage.googleapis.com/golang/go1.8.1.linux-amd64.tar.gz
```

(3) 解压压缩文件go1.8.1.linux-amd64.tar.gz到/usr/local/目录下，安装Go。

```
sudo tar -C /usr/local -xzf go1.8.1.linux-amd64.tar.gz
```

(4) 配置Go环境变量。

```
mkdir $HOME/go
sudo vim /etc/profile
```

在/etc/profile文件中的判断语句下方添加下面的Go环境变量。

```
# go env
export PATH=$PATH:/usr/local/go/bin
export GOPATH=$HOME/go
```

修改后的profile文件如下所示：

```
if [ -d /etc/profile.d ]; then
    for i in /etc/profile.d/*.sh; do
        if [ -r $i ]; then
            . $i
        fi
    done
unset i
fi
```

```
export PATH=$PATH:/usr/local/go/bin
export GOPATH=$GOPATH/go
```

执行source命令使其生效，然后使用go env命令输出Go环境配置信息。

```
source /etc/profile
go env
```

(5) 测试Go语言。

输入以下命令：

```
cd $GOPATH
mkdir -p src/hello
cd src/hello/
vim hello.go
```

输入测试代码：

```
package main
import "fmt"
func main() {
    fmt.Printf("hello, world\n")
}
```

保存退出，使用go build命令编译后执行，显示如下结果则代表Go环境配置成功。

```
→ go build
→ ls
hello hello.go
→ ./hello
hello, world
```


2. Docker环境

Docker是一个开源的应用容器引擎,可以让开发者将应用以及依赖包打包到一个可移植的容器中,然后发布到任何流行的Linux机器上,也可以实现虚拟化。容器是完全使用沙箱机制,相互之间不会有任何接口。

Docker是一个程序运行、测试、交付的开放式平台,它的设计目的就是让使用者可以快速地与应用进行交互。在Docker中,开发人员可以将程序分为不同的基础部分,每一个基础部分都可以当作一个应用程序来管理。Docker能够帮助开发者快速地测试、快速地编码、快速地交付,并且缩短从编码到运行应用的周期。

本章后面的Hyperledger Fabric的开发实战是使用Docker容器来管理开发的,因此需要先在Ubuntu操作系统上配置好Docker运行环境。

Docker为Ubuntu提供了快速的安装方式,只需以下几行命令,就可以快速地安装在虚拟操作系统中。

(1) 添加远程仓库地址。

```
sudo apt-get -y install \
    apt-transport-https \
    ca-certificates \
    curl
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
sudo apt-get update
```

(2) 安装docker-ce。

```
sudo apt-get -y install docker-ce
```

(3) 测试Docker。

```
sudo docker run hello-world
```

如果显示以下结果,则表示Docker安装成功。

```
→ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

在Docker安装完成之后,还需要额外安装docker-compose工具,从而可以通过配置文件来部署与启动容器,docker-compose安装使用如下命令。

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.11.2/docker-compose-$(uname
-s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

查看Docker和docker-compose的版本，如显示以下结果，则Docker环境安装完成。

```
→docker -v
Docker version 17.03.1-ce, build c6d412e
→docker-compose -v
docker-compose version 1.11.2, build dfed245
```

由于国内无法访问Docker官方站点，可以访问一些镜像站，比如阿里云、CloudDao等，通过下面的命令设置CloudDao配置Docker加速器，快速下载Docker镜像。

```
curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh -s http://176a5be3.m.daocloud.io
sudo systemctl restart docker.service
```

5.2 链码开发指南

链码是部署在Hyperledger Fabric网络节点上、可与分布式账本进行交互的一段程序代码，即狭义范畴上的“智能合约”。本节将介绍如何在Fabric上开发链码。

5.2.1 实现智能合约的接口

5

Hyperledger支持多种计算机语言实现的链码，包括Golang、JavaScript、Java等。Go语言主要依赖链码的shim接口来实现核心业务逻辑。Fabric 1.0的shim接口主要包含两个核心的函数，分别是Init和Invoke。功能函数都以函数名和字符串结构作为输入，主要的区别在于函数的功能用途。

(1) Init函数

当首次部署链码时，Init函数会被调用。该函数用来做一些初始化的工作。

(2) Invoke函数

当通过调用链码来做一些实际性的工作时，可以使用Invoke函数。发起的交易将会被链上的区块获取并记录。

它以被调用的函数名作为参数，并基于该参数调用链码中匹配的Go函数。

Fabric 1.0会将Query函数整合进Invoke函数，以便灵活进行查询和调用。

(3) Main函数

最后，需要创建一个Main函数，当每个节点部署链码的实例时，该函数会被调用，不过仅仅当链码在某节点上注册时会被调用。

5.2.2 智能合约的依赖关系

开发链码需要引入如下的软件包。

- `fmt`: 包含了`Println`等标准函数。
- `errors`: 标准的`errors`类型包。
- `github.com/hyperledger/fabric/core/chaincode/shim`: 与链码节点交互的接口代码。`shim`包提供了`stub.PutState()`与`stub.GetState()`等方法来写入和查询区块链上键/值的状态。核心的`shim`包, 通过封装gRPC消息的方式到验证节点来完成操作。

5.2.3 智能合约的数据格式

在Fabric 0.6中, 数据持久化存储采用`rocksdb`, 除了基本的键/值格式的数据支持之外, 智能合约的Go接口支持以`Table`的形式定义合约中的数据, 但是Fabric 1.0中移除了`Table`相关接口。

在Fabric 1.0中, 建议基于JSON格式构造智能合约数据, 在例子`marbles02`中, 可定义结构体`marble`为JSON格式:

```
type marble struct {
    ObjectType    string `json:"docType"`
    Name          string `json:"name"`
    Color         string `json:"color"`
    Size          int    `json:"size"`
    Owner         string `json:"owner"`
}
```

该结构体定义了`marble`的类型、名称、颜色、尺寸、拥有者等属性。

表5.1所示是`Table`实现方式和JSON实现方式的一些对比。

表5.1 Table实现方式和JSON实现方式对比

基于Table的实现方式	基于JSON的实现方式
<ul style="list-style-type: none"> ● 关系型数据库的方式 ● 需要之前定义好表结构 ● 在之后合约的升级过程中很难更改表结构 ● 不支持分层数据 ● 和账本的底层数据不一致 ● 该方式无法满足Fabric的功能, 例如无法像预期的那样随意按列查询数据 ● 代码结构更复杂, 导致合约代码复杂 	<ul style="list-style-type: none"> ● NoSQL的方式: <code>key/value</code>(<code>LevelDB</code>), <code>documentdb</code>(<code>CouchDB</code>) ● 不需要事先定义好表结构 ● 在以后的合约升级过程中可以很容易地添加新的JSON域 ● 支持分层数据 ● 和账本的底层数据一致 ● 该方式可以满足Fabric的功能, 查询基于<code>key</code>或者<code>partial key range</code> ● 更少的代码量, 采用内置结构JSON marshaling ● 更适合下一代的账本功能要求: <ul style="list-style-type: none"> - 可以以任何字段查询账本, 无论在合约内还是合约外 - 在使用基于JSON的数据库时更加有效 (<code>CouchDB</code>)

5.2.4 智能合约的接口解析

在Fabric 1.0中, 移除了Table相关的所有读写接口, 同时也提供了如下读写接口供合约调用。

- ❑ `GetState(key string) ([]byte, error)`: 按照给定的key查询返回对应值的byte数组。
- ❑ `PutState(key string, value []byte) error`: 将给定的key和value写入账本。
- ❑ `DelState(key string) error`: 在账本中移除给定的key和对应的value记录。
- ❑ `CreateCompositeKey(objectType string, attributes []string) (string, error)`: 该接口会构建一个由objectType和attributes组成的组合key, 其中objectType和attributes必须是有效的utf8字符串且不能包含string(0)和string(utf8.MaxRune)。
- ❑ `SplitCompositeKey(compositeKey string) (string, []string, error)`: 该接口会将构建好的组合key重新按照组合方式拆分成objectType和attributes。
- ❑ `PartialCompositeKeyQuery(objectType string, keys []string) (StateQueryIteratorInterface, error)`: 在组合键中查询包含给定objectType和keys的组合键, 返回这些组合键的迭代器, 注意keys必须是组合键的一部分, 如果是全部组合键的话会返回空的迭代器。
- ❑ `RangeQueryState(startKey, endKey string) (StateQueryIteratorInterface, error)`: 在账本中查询key值在startKey和endKey之间的记录的迭代器, 迭代器中key的顺序是随机的。
- ❑ `GetQueryResult(query string) (StateQueryIteratorInterface, error)`: 该接口会在状态数据库(如CouchDB)中执行rich query, 实现对任意字段的查询, 参数query是底层状态数据库的查询语法, 该接口仅支持使用状态数据库(如CouchDB)的情况。返回一个包含所需记录的迭代器。

5

5.2.5 智能合约案例代码分析

本智能合约案例基于Fabric 1.0实现资产(弹珠)的创建、转让和查询, 旨在让开发者学习智能合约的基础知识, 并学会在Fabric网络上开发一些应用。

该案例的主要功能如下。

- ❑ 管理员可以给用户创建一种弹珠(资产), 并将其存储在该智能合约的账本中。
- ❑ 管理员可以读取和查询存储在该智能合约账本上的所有弹珠(资产)信息。
- ❑ 管理员可以将一位用户一种弹珠(资产)转移给另一位用户。
- ❑ 管理员可以删除存储在该智能合约账本上的弹珠(资产)。

目前Fabric 1.0将query()方法合并至Invoke接口, 统一了使用方式。智能合约本身采用键值对的形式存储数据, 我们在本例中为了存储更加复杂的数据结构, 采用了JSON对象的形式。

1. 将JSON数据写入账本

可通过如下命令将指定的key和JSON数据写入账本, 其中JSON结构体的定义在第三部分中已经介绍。


```
// 创建 marble 对象并转为JSON格式
objectType := "marble"
marble := &marble{objectType, marbleName, color, size, owner}
marbleJSONAsBytes, err := json.Marshal(marble)
if err != nil {return shim.Error(err.Error())}
// 存储marble对象到state世界状态
err = stub.PutState(marbleName, marbleJSONAsBytes)
if err != nil {return shim.Error(err.Error())}
```

其中，第3行实现了定义的JSON结构体，第4行以byte数组的形式存储JSON数据，第7行调用Putstate接口，将name作为key，JSON数据作为value写入账本。

2. 按照查询需求创建索引（不使用CouchDB等状态数据库）

在本例中，如果有按照颜色查询Marble的需求，则需按如下方法建立索引：

```
// 输入索引key
indexName := "color~name"
colorNameIndexKey, err := stub.CreateCompositeKey(indexName, []string{marble.Color,
marble.Name})
// 异常处理
if err != nil {return shim.Error(err.Error())}
value := []byte{0x00}
// 更新状态变量
stub.PutState(colorNameIndexKey, value)
```

其中，第2行创建了由color和name组成的组合键，建立了color和name（主键）之间的索引关系，索引以名称“color~name”区别于其他索引。第9行将这个组合键作为key，一个空character作为value写入账本，索引信息在key中。

3. 按照颜色查询（不使用CouchDB等状态数据库）

查询命令如下：

```
// 根据颜色查询Marbles
coloredMarbleResultsIterator, err := stub.PartialCompositeKeyQuery("color~name",
[]string{color})
// 异常处理
if err != nil {return shim.Error(err.Error())}
defer coloredMarbleResultsIterator.Close()
// 迭代遍历查询到的marbles结果
var i int
for i = 0; coloredMarbleResultsIterator.HasNext(); i++ {
    colorNameKey, _, err := coloredMarbleResultsIterator.Next()
    if err != nil { return shim.Error(err.Error()) }
}
// 通过key取值
objectType, compositeKeyParts, err := stub.SplitCompositeKey(colorNameKey)
if err != nil {return shim.Error(err.Error())}
returnedColor := compositeKeyParts[0]
returnedMarbleName := compositeKeyParts[1]
// 输出查到的marble信息
fmt.Printf("- found a marble from index:%s color:%s name:%s\n",
```

```
objectType, returnedColor, returnedMarbleName)
marbleAsBytes, err := stub.GetState(returnedMarbleName)
```

其中，第2行调用PartialCompositeKeyQuery接口查出包含符合颜色要求的组合键的迭代器（该函数调用了RangeQueryState接口），然后对迭代器做循环，取出符合要求的组合键，再用SplitCompositeKey接口解析出marble的name（主键），最后用取出的name（主键）查出符合颜色要求的marble的JSON数据。

4. 按照拥有者查询（使用CouchDB等状态数据库）

使用CouchDB的rich query功能按照拥有者查询，在该情况下无需在合约里额外建立任何索引信息，可以直接用CouchDB的查询语法进行任意查询：

```
// 根据所拥有者查询Marbles对象
func (t *SimpleChaincode) queryMarblesByOwner(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
    // 参数格式不对
    if len(args) < 1 {
        return shim.Error("Incorrect number of arguments. Expecting 1")
    }
    owner := strings.ToLower(args[0])
    queryString := fmt.Sprintf("{\"selector\":{\"docType\":\"marble\",\"owner\":\"%s\"}}",
owner)
    queryResults, err := getQueryResultForQueryString(stub, queryString)
    // 异常处理
    if err != nil {return shim.Error(err.Error())}
    // 返回结果
    return shim.Success(queryResults)
}
```

其中，第9行的查询语句为CouchDB的查询语法，具体用法可以参考：<http://docs.couchdb.org/en/2.0.0/api/database/find.html#selector-syntax>。

getQueryResultForQueryString函数定义如下所示。

```
func getQueryResultForQueryString(stub shim.ChaincodeStubInterface, queryString string) ([]byte,
error) {
    // 取得结果迭代器
    resultsIterator, err := stub.GetQueryResult(queryString)
    if err != nil {return nil, err}
    defer resultsIterator.Close()
    // buffer是一个包含了查询结果的JSON数组
    var buffer bytes.Buffer
    buffer.WriteString("[")
    bArrayMemberAlreadyWritten := false
    // 迭代遍历结果
    for resultsIterator.HasNext() {
        queryResultKey, queryResultRecord, err := resultsIterator.Next()
        // 异常处理
        if err != nil {return nil, err}
        if bArrayMemberAlreadyWritten == true {
            buffer.WriteString(",")
```

```
}  
// 构造JSON串  
buffer.WriteString("{\"Key\":")  
buffer.WriteString("\"")  
buffer.WriteString(queryResultKey)  
buffer.WriteString("\")  
buffer.WriteString(", \"Record\":")  
buffer.WriteString(string(queryResultRecord))  
buffer.WriteString("}")  
bArrayMemberAlreadyWritten = true  
}  
buffer.WriteString("]")  
fmt.Printf("- getQueryResultForQueryString queryResult:\n%s\n", buffer.String())  
return buffer.Bytes(), nil  
}
```

该函数调用了GetQueryResult接口，循环返回的迭代器，循环迭代器中的内容，将查询出的key和value构建为JSON数组，每次迭代以逗号分隔，最终将全部结果以byte数组的形式返回。

5.3 CLI 应用实例

当前，Fabric支持两种模式的应用开发：CLI和SDK接口。CLI是Command Line Interface的缩写，即命令行接口。SDK则是Software Development Kit的缩写，即软件开发工具包。本节针对CLI接口介绍如何使用命令行的方式来开发部署以及调用链码。

5.3.1 CLI 介绍

通过CLI命令行接口，用户可以简单地完成与区块链的交互。目前，Fabric支持的命令接口如表5.2所示。

表5.2 Fabric支持的命令接口

命令行参数	功 能	命令行参数	功 能
version	查看版本信息	chaincode invoke	调用链码
node start	启动节点	chaincode query	查询链码
node status	查看节点状态	channel create	创建通道
node stop	停止节点	channel join	加入通道
chaincode deploy	部署链码		

通过以下命令，可以查询更多CLI命令相关的信息。

```
# 启动网络后进入cli容器  
docker exec -it cli bash  
# 进入cli容器后运行peer命令  
peer
```

Peer命令参数如下所示:

Usage:

```
peer [flags]
peer [command]
```

Available Commands:

```
chaincode  chaincode specific commands.
channel     channel specific commands.
logging     logging specific commands.
node        node specific commands.
version     Print fabric peer version.
```

Flags:

```
-h, --help                help for peer
--logging-level string     Default logging level and overrides
--test.coverprofile string Done (default "coverage.cov")
-v, --version             Display current vision of fabric peer server
```

Use "peer [command] -help" for more information about a command

5.3.2 CLI 应用开发

前面已经对Fabric提供的CLI命令行进行了介绍说明, 本节则将通过/fabric/examples/e2e_cli/目录下的例子来学习CLI命令行接口的使用。

1. 准备工作

(1) 拉取docker镜像

在目录下, 存在一个名为download-dockerimages.sh的脚本文件, 可以直接运行帮助用户拉取v1.0 alpha版本的镜像, 运行脚本直接将v1.0 alpha版本的镜像下载到本地。

```
cd examples/e2e_cli/
# 将脚本文件更改权限为可执行
chmod +x download-dockerimages.sh
# 执行拉取镜像脚本
sudo ./download-dockerimages.sh ./download-dockerimages.sh
```

执行完成后会显示一个下载的镜像列表, 结果如下所示。

```
===> List out Hyperledger docker images
Hyperledger/fabric-ca          latest          35311d8617b4   4 weeks ago   240MB
Hyperledger/fabric-ca          x86_64-1.0.0-alpha 35311d8617b4   4 weeks ago   240MB
Hyperledger/fabric-couchdb     latest          35311d8617b4   4 weeks ago   1.51GB
Hyperledger/fabric-couchdb     x86_64-1.0.0-alpha 35311d8617b4   4 weeks ago   1.51GB
Hyperledger/fabric-kafka       latest          35311d8617b4   4 weeks ago   1.3GB
Hyperledger/fabric-kafka       x86_64-1.0.0-alpha 35311d8617b4   4 weeks ago   1.3GB
Hyperledger/fabric-zookeeper   latest          35311d8617b4   4 weeks ago   1.31GB
```


Hyperledger/fabric-zookeeper	x86_64-1.0.0-alpha	35311d8617b4	4 weeks ago	1.31GB
Hyperledger/fabric-order	latest	35311d8617b4	4 weeks ago	182MB
Hyperledger/fabric-order	x86_64-1.0.0-alpha	35311d8617b4	4 weeks ago	182MB
Hyperledger/fabric-peer	latest	35311d8617b4	4 weeks ago	184MB
Hyperledger/fabric-peer	x86_64-1.0.0-alpha	35311d8617b4	4 weeks ago	184MB
Hyperledger/fabric-javaenv	latest	35311d8617b4	4 weeks ago	1.42GB
Hyperledger/fabric-javaenv	x86_64-1.0.0-alpha	35311d8617b4	4 weeks ago	1.42GB
Hyperledger/fabric-ccenv	latest	35311d8617b4	4 weeks ago	1.29GB
Hyperledger/fabric-ccenv	x86_64-1.0.0-alpha	35311d8617b4	4 weeks ago	1.29GB

也可以直接通过docker命令手动拉取全部镜像文件, 命令如下:

```
docker pull hyperledger/fabric-orderer:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-peer:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-zookeeper:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-couchdb:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-kafka:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-ca:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-ccenv:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-javaenv:x86_64-1.0.0-alpha
```

(2) 构建Configtxgen工具

Configtxgen是一个官方用于在Fabric中形成创世区块规则和通道配置文件的工具。构建方法很简单, 进入项目主目录, 用make命令即可。

```
cd $GOPATH/src/github.com/hyperledger/fabric
# 构建configtxgen工具
make configtxgen
```

构建成功后会创建一个configtxgen二进制文件, 位于build/bin/configtxgen目录下, 终端则会打印如下信息。

```
→make configtxgen
build/bin/configtxgen
CGO_FLAGS="" GOBIN=/home/blockchain/go/src/github.com/Hyperledger/fabric/build/bin
ata.Version=1.0.0-snapshot-d5d1293 -X github.com/Hyperledger/fabric/common/metadata.BaseVersion
rg.hyperledger.fabric -X github.com/Hyperledger/fabric/common/metadata.DockerNamespace=hyperledger
Hyperledger" github.com/Hyperledger/fabric/common/configtx/tool/configtxgen
Binary available as build/bin/configtxgen
```

2. 编写代码

在本实例中, 需要编写3个代码文件, 其中包含两个YAML配置文件和一个Go语言链码文件。其中, configtx.yaml用于配置初始区块和通道的规则, 本实例中定义了一个包含1个Orderer节点和4个Peer节点的Fabric区块链网络。docker-compose.yaml则是docker-compose工具的配置文件, 用于配置容器间的各种属性和操作。它们的详细说明如下。

(1) configtx.yaml

这一配置文件定义了实例网络的通道规则，运用前面构建的Configtxgen工具可以生成创世区块和通道配置文件

```
---
Profiles:
  TwoOrgs:
    Orderer:
<<: *OrdererDefaults
      Organizations:
        - *OrdererOrg
      Application:
<<: *ApplicationDefaults
      Organizations:
        - *Org0
        - *Org1
# 定义两个组Org0和Org1
Organizations:
  - &Org0
    Name: Org0MSP
    ID: Org0MSP
    # 成员服务路径
    MSPDir: examples/e2e_cli/crypto/peer/peer0/localMspConfig
    # BCCSP (Blockchain crypto provider):选择提供的加密算法
    BCCSP:
      Default: SW
      SW:
        Hash: SHA2
        Security: 256
        FileKeyStore:
          KeyStore:
            AnchorPeers:
              - Host: peer0
                Port: 7051
        - &Org1
    ...
# 定义与共识服务相关的信息
Orderer: &OrdererDefaults
  # 共识类型目前可选 solo 或者 Kafka
  OrdererType: solo
  Addresses:
    - orderer0:7050
  BatchTimeout: 2s
  BatchSize:
    MaxMessageCount: 10
    AbsoluteMaxBytes: 99 MB
    PreferredMaxBytes: 512 KB
  Kafka:
    Brokers:
      - 127.0.0.1:9092
  Organizations:
Application: &ApplicationDefaults
Organizations:
```

(2) docker-compose.yml

```

version: '2'
services:
  # 本网络中唯一的共识节点orderer0, 用于提供ordering service
  orderer0:
    container_name: orderer0
    image: hyperledger/fabric-orderer
    # [环境变量, 此处略]
    environment:
      # [详细请查看文件]
    # 工作路径
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric
    command: orderer
    volumes:
      - ./crypto/orderer:/var/hyperledger/orderer
    # 端口
    ports:
      - 7050:7050
  # 4个peer节点的配置信息
  peer0:
    container_name: peer0
    extends:
      file: peer-base/peer-base.yml
      service: peer-base
    # [环境变量, 此处略]
    environment:
      # [详细请查看文件]
    # 磁碟区
    volumes:
      - /var/run:/host/var/run/
      - ./crypto/peer/peer0/localMspConfig:/etc/hyperledger/fabric/msp/sampleconfig
    # 端口
    ports:
      - 7051:7051
      - 7053:7053
    # 依赖
    depends_on:
      - orderer0
  peer1:
    ...
  peer2:
    ...
  peer3:
    ...
  # cli容器, 用于提供命令行操作环境
  cli:
    container_name: cli
    image: hyperledger/fabric-peer
    tty: true
    # [环境变量, 此处略]
    environment:
      # [详细请查看文件]
    # 工作目录

```

```

working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
# 运行脚本
command: /bin/bash -c './scripts/script.sh ${CHANNEL_NAME};'
# 磁盘区
volumes:
    - /var/run/:/host/var/run/
    - ./examples/:/opt/gopath/src/github.com/hyperledger/fabric/examples/
    - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/examples/chaincode/go
    - ./crypto:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
    - ./scripts:/opt/gopath/src/github.com/hyperledger/fabric/peer/scripts/
# 依赖
depends_on:
    - orderer0
    - peer0
    - peer1
    - peer2
    - peer3

```

(3) example_chaincode02.go

```

package main
// 需要的依赖包
import (.....)
// SimpleChaincode实现链码接口
type SimpleChaincode struct {}
// Init接口实现, 用于初始化
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Init")
    _, args := stub.GetFunctionAndParameters()
    var A, B string // 实体A,B
    var Aval, Bval int // 实体对应的值
    var err error // 错误
    # 参数太多
    if len(args) != 4 {
        return shim.Error("Incorrect number of arguments. Expecting 4")
    }
    // 实例化链码对象, A赋值操作
    A = args[0]
    Aval, err = strconv.Atoi(args[1])
    if err != nil {
        return shim.Error("Expecting integer value for asset holding")
    }
    // 实例化链码对象, B赋值操作
    B = args[2]
    Bval, err = strconv.Atoi(args[3])
    if err != nil {
        return shim.Error("Expecting integer value for asset holding")
    }
    fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)
    // 将A写入到状态变量中
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    if err != nil {
        return shim.Error(err.Error())
    }
}

```



```

}
// 将B写入到状态变量中
err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
if err != nil {
    return shim.Error(err.Error())
}
return shim.Success(nil)
}

// Invoke实现方法
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Invoke")
    function, args := stub.GetFunctionAndParameters()
    if function == "invoke" {
        // 调用invoke方法
        return t.invoke(stub, args)
    } else if function == "delete" {
        // 调用idelete方法
        return t.delete(stub, args)
    } else if function == "query" {
        // 调用idelete方法
        return t.query(stub, args)
    }
    return shim.Error("Invalid invoke function name. Expecting \"invoke\" \"delete\" \"query\"")
}

// invoke方法
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var A, B string // 实体A,B
    var Aval, Bval int // A,B对应的值
    var X int // 交易需要转移的值
    var err error // 错误
    if len(args) != 3 {
        return shim.Error("Incorrect number of arguments. Expecting 3")
    }
    // 赋值A,B
    A = args[0]
    B = args[1]
    // 获取A的状态变量
    Avalbytes, err := stub.GetState(A)
    if err != nil { return shim.Error("Failed to get state") }
    if Avalbytes == nil { return shim.Error("Entity not found") }
    Aval, _ = strconv.Atoi(string(Avalbytes))
    Bvalbytes, err := stub.GetState(B)
    if err != nil { return shim.Error("Failed to get state") }
    if Bvalbytes == nil { return shim.Error("Entity not found") }
    Bval, _ = strconv.Atoi(string(Bvalbytes))
    // 执行调用
    X, err = strconv.Atoi(args[2])
    if err != nil { return shim.Error("Invalid transaction amount, expecting a integer value") }
    // 值操作, A减少, B增加
    Aval = Aval - X
    Bval = Bval + X
    fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)
    // 将A写入状态变量
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))

```

```

if err != nil { return shim.Error(err.Error()) }
// 将B写入状态变量
err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
if err != nil { return shim.Error(err.Error()) }
return shim.Success(nil)
}
// 删除状态变量方法
func (t *SimpleChaincode) delete(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    if len(args) != 1 { return shim.Error("Incorrect number of arguments. Expecting 1") }
    A := args[0]
    // 根据key删除状态变量
    err := stub.DelState(A)
    if err != nil { return shim.Error("Failed to delete state") }
    return shim.Success(nil)
}
// 根据key查询值查询链码方法
func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var A string // Entities
    var err error
    if len(args) != 1 { return shim.Error("Incorrect number of arguments. Expecting name of the
    person to query") }
    A = args[0]
    // 获取A的状态变量
    Avalbytes, err := stub.GetState(A)
    if err != nil {
        jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
        return shim.Error(jsonResp)
    }
    if Avalbytes == nil {
        jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"
        return shim.Error(jsonResp)
    }
    // JSON格式响应
    jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) + "\"}"
    fmt.Printf("Query Response:%s\n", jsonResp)
    return shim.Success(Avalbytes)
}
// 主函数
func main() {
    err := shim.Start(new(SimpleChaincode))
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}

```

3. 启动网络与链码调用

(1) 生成创世区块和通道配置文件

根据相应的配置文件，用户可以使用Configtxgen工具，通过下面的命令生成创世区块order.block和通道配置文件channel.tx。

```

# 当前应位于根目录
# 使用Configtxgen工具生成创世区块order.block

```

```
./build/bin/configtxgen -profile TwoOrgs -outputBlock orderer.block
mv orderer.block examples/e2e_cli/crypto/orderer/orderer.block
# 创建通道配置channel.tx
./build/bin/configtxgen -profile TwoOrgs -outputCreateChannelTx channel.tx -channelID mychannel
mv channel.tx examples/e2e_cli/crypto/orderer/channel.tx
```

(2) 脚本自动启动网络并执行指定操作

在官方代码的例子examples/e2e_cli/目录中，提供了一个脚本执行，包括创建通道、部署链码等所有操作。执行docker-compose命令即可完成一站式的体验。使用以下命令启动网络。

```
# 进入例子目录
cd examples/e2e_cli/
# 启动网络，注意通道名要与前面生成配置信息时相同
CHANNEL_NAME=mychannel docker-compose -f docker-compose.yaml up -d
# 动态打印日志
docker logs -f cli
```

运行之后，终端会动态地打印所有操作的信息，如果最后显示如下结果，则代表运行成功。

```
2017-04-10 13:47:19:225 UTC [logging] InitFromViper -> DEBU 001 Setting default logging level to
DEBUG for command 'chaincode'
2017-04-10 13:47:19:228 UTC [map] GetLocalMSP -> DEBU 002 Return
ing existing local MSP
2017-04-10 13:47:19:228 UTC [map] GetDefaultSigningIdentity-> DE
BU 003 Obtaining default signing identity
2017-04-10 13:47:19:228 UTC [map] Sign -> DEBU 004 Sign: plaintext
:0A8F050A59080322096D796368616E6E...6D7963631A0A0A0571756572790A02017-04-10 13:47:19:228 UTC
[map] Sign -> DEBU 005 Sign: digest:
AC78351F08FFD681EF3D1F853E1601DA0859D66E18C752908CE18A6F0F65E14
Query Result: 90
2017-04-10 13:47:19:228 UTC [map] main -> INFO 006 Exiting...
=====Query on PEER# on channel 'mychannal' is successful=====
=====All GOOD, End-2-End execution completed=====
```

(3) 手动执行交易

前面通过一个写好的脚本执行了所有的操作，接下来需要手动进行以上操作，比如创建通道、加入通道、安装链码、调用链码等。要开始手动执行操作，首先得将前面所启动的网络关掉，examples/e2e_cli/中提供了便捷方式，直接使用一条命令即可清理整个网络和容器。

```
./network_setup.sh down
```

下面将通过命令手动进行上面的操作，首先需要在docker-compose.yaml命令中将script.sh脚本注释掉，不使用它来执行操作，如下所示。

```
2017-04-10 13:47:19:225 UTC [logging] InitFromViper -> DEBU 001 Setting default logging level to
DEBUG for command 'chaincode'
2017-04-10 13:47:19:228 UTC [map] GetLocalMSP -> DEBU 002 Return
ing existing local MSP
2017-04-10 13:47:19:228 UTC [map] GetDefaultSigningIdentity-> DE
BU 003 Obtaining default signing identity
2017-04-10 13:47:19:228 UTC [map] Sign -> DEBU 004 Sign: plaintext
```

```

:0A8F050A59080322096D796368616E6E...6D7963631A0A0A0571756572790A02017-04-10 13:47:19:228 UTC
[map] Sign -> DEBU 005 Sign: digest:
AC78351F08FFD681EF3D1F853E1601DA0859D66E18C752908CE18A6F0F65E14
Query Result: 90
2017-04-10 13:47:19:228 UTC [map] main -> INFO 006 Exiting...
=====Query on PEER# on channel 'mychannel' is successful=====
=====All GOOD, End-2-End execution completed=====
    working_dir:/opt/gopath/src/github.com/Hyperledger/fabric/peer
    #commond: /bin/bash -c './sscripts/script.sh ${CHANNEL_NAME}';'
    #commond: /bin/bash
    Volumes:
        /var/run/:/host/var/run/

```

重新启动网络，在启动前需要先清理开始的网络。部署和调用步骤如下。

(1) 运用docker-compose命令启动网络。

```
CHANNEL_NAME=mychannel docker-compose -f docker-compose.yaml up -d
```

(2) 进入cli控制台。

```
docker exec -it cli bash
```

(3) 设置orderer0的全局环境变量。

```

#定义节点的成员服务对应的数字签名路径 CORE_PEER_MSPCONFIGPATH
CORE_PEER_MSPCONFIGPATH=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/orderer/localMspConfig
# 定义成员服务的标识CORE_PEER_LOCALMSPID
CORE_PEER_LOCALMSPID="OrdererMSP"
# 定义CA的路径ORDERER_CA
ORDERER_CA=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/orderer/localMspConfig/cacerts/ordererOrg0.pem

```

(4) 创建通道，orderer0使用peer channel create命令创建了一个mychannel通道。

```

peer channel create -o orderer0:7050 -c mychannel -f crypto/orderer/channel.tx --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA

```

(5) Peer节点加入mychannel通道，在每次使用peer channel join命令时，需要通过配置全局环境变量来指定需要加入的Peer节点。

```

# 设置peer0环境变量，指定当前以peer0为操作对象
CORE_PEER_MSPCONFIGPATH=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/peer/peer0/localMspConfig
CORE_PEER_ADDRESS=peer0:7051
CORE_PEER_LOCALMSPID=Org0MSP
CORE_PEER_TLS_ROOTCERT_FILE=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/peer/peer0/localMspConfig/cacerts/peerOrg0.pem
# peer0加入channel
peer channel join -b mychannel.block
# 设置peer1环境变量，指定当前以peer1为操作对象
CORE_PEER_MSPCONFIGPATH=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/peer/peer1/localMspConfig
CORE_PEER_ADDRESS=peer1:7051

```



```

CORE_PEER_LOCALMSPID="Org0MSP"
CORE_PEER_TLS_ROOTCERT_FILE=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/peer/peer1/
localMspConfig/cacerts/peerOrg0.pem
# peer1加入channel
peer channel join -b mychannel.block
# 设置peer2环境变量, 指定当前以peer2为操作对象
CORE_PEER_MSPCONFIGPATH=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/peer/peer2/loca
lMspConfig
CORE_PEER_ADDRESS=peer2:7051
CORE_PEER_LOCALMSPID=Org1MSP
CORE_PEER_TLS_ROOTCERT_FILE=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/peer/peer2/
localMspConfig/cacerts/peerOrg1.pem
# peer2加入channel
peer channel join -b mychannel.block
# 设置peer3环境变量, 指定当前以peer3为操作对象
CORE_PEER_MSPCONFIGPATH=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/peer/peer3/loca
lMspConfig
CORE_PEER_ADDRESS=peer3:7051
CORE_PEER_LOCALMSPID=Org1MSP
CORE_PEER_TLS_ROOTCERT_FILE=$GOPATH/src/github.com/hyperledger/fabric/peer/crypto/peer/peer3/
localMspConfig/cacerts/peerOrg1.pem
# peer3加入channel
peer channel join -b mychannel.block

```

(6) 在所有Peer节点加入channel之后, 就可以指定Peer节点安装链码, 安装链码的命令为peer chaincode install。

```

# 设置链码路径变量, 方便调用
CHAINCODE_DIR=github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02

```

在Peer0和Peer2上安装链码。

```

# 首先依照上面的方法设置peer0和peer2的环境变量, 因为相同, 此处省略, 后面都以setEnv + peer节点
# 设置Env 0
peer chaincode install -n mycc -v 1.0 -p $CHAINCODE_DIR
# 设置Env 2
peer chaincode install -n mycc -v 1.0 -p $CHAINCODE_DIR

```

(7) 在Peer节点安装成功后, 就可以在对应节点上实例化链码, 取得链码对象。下面先在Peer2节点上实例化链码对象mycc, 同时携带参数初始化a和b的值, 并指定背书策略为"OR('Org0MSP.member', 'Org1MSP.member')"

```

# setEnv 2
peer chaincode instantiate -o orderer0:7050 --tls $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA -C
mychannel -n mycc -v 1.0 -p $CHAINCODE_DIR -c '{"Args":["init","a","100","b","200"]}' -P
"OR('Org0MSP.member','Org1MSP.member')"

```

(8) 在Peer0节点上查询a的值, 会得到结果为100。

```

# setEnv 0
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'

```

(9) 在Peer0上发送调用交易, 调用invoke方法从a转移10到b。

```
peer chaincode invoke -o orderer0:7050 --tls $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA -C mychannel -n mycc -c '{"Args":["invoke","a","b","10"]}'
```

(10) 在Peer3上安装链码mycc。

```
setEnv 3
peer chaincode install -n mycc -v 1.0 -p $CHAINCODE_DIR
```

(11) 在Peer3上调用mycc查询方法查询返回的值，取得结果为90，正确。

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
```

(12) 程序执行完成。

5.4 SDK 应用实例

Hyperledger Fabric SDK为开发人员提供了一个结构化的库环境，用于编写和测试链码应用程序。Fabric提供的SDK是完全可配置的，并可通过标准接口进行扩展。SDK API使用基于gRPC的协议缓冲区（protocol buffer）提供交易处理、成员服务管理和节点遍历等功能。本节将重点介绍如何利用SDK API开发基于Fabric的区块链应用。

5.4.1 SDK 介绍

Hyperledger Fabric SDK客户端有多种实现，当前包括Go、Node.js、Java以及Python这4种。本实例以Node.js为例进行介绍，Node.js的Hyperledger Fabric SDK是面向对象的编程风格设计，其模块化使得应用程序开发人员可以自己基于基础API插入核心函数的实现，比如加密算法、state的持久存储和日志记录。Hyperledger Fabric Node.js SDK客户端提供的API接口可以分为两类：fabric-ca-client和fabric-client。

fabric-ca-client负责和fabric-ca组件进行交互，提供成员管理服务。其提供的方法如表5.3所示。

表5.3 fabric-ca-client接口

命令行参数	功 能
NewFabricCAClient()	新建Fabric客户端
Enroll(enrollmentID, enrollmentSecret)	登记一个注册的用户
Reenroll(user)	登记一个已经登记过的用户
Register(register, request)	注册
Revoke(registrar, request)	撤销证书
createSigningIdentity(user)	创建一个签名身份

fabric-client负责同Hyperledger Fabric的核心组件进行交互，如Peer节点、Orderer节点和事件流。其提供的部分接口如表5.4所示。

表5.4 fabric-client接口

命令行参数	功 能
newChain(name)	新建一条链
getChain(name)	通过name获取链
newPeer(url, opts)	新建Peer节点
newOrderer(url, opts)	新建Orderer节点
newMSP(msp_def)	新建成员服务
createChannel(request)	创建通道
updateChannel(request)	更新通道
queryChainInfo(name, peers)	查询链信息
queryChannels(peer)	查询Peer节点加入的通道
queryInstalledChaincodes(peer)	查询Peer节点上安装的链码
installChaincode(request)	安装链码
setStateStore(keyValueStore)	设置键值存储的
saveUserToStateStore()	保存用户到State
setUserContext(user, skipPersistence)	设置用户上下文
getUserContext(name, checkPersistence)	获取用户上下文
loadUserFromStateStore(name)	从State中加载用户
getStateStore()	获取State
buildTransactionID(nonce, userContext)	构建交易
createUser(opts)	创建用户
setLogger(logger)	记录日志
setMSPManager(msp_manager)	设置成员服务管理
getMSPManager()	获取当前的成员服务管理
addPeer(peer)	添加Peer
removePeer(peer)	移除Peer
getPeers()	获取Peer节点集合
addOrderer(orderer)	添加Orderer
removeOrderer(orderer)	移除Orderer
getGenesisBlock(request)	获取创世区块
joinChannel(request)	加入通道
queryBlockByHash(blockHash)	查询区块
queryBlock(blockNumber)	查询区块
queryTransaction(transactionID)	查询交易
queryInstantiatedChaincodes()	查询实例化的链码
sendTransactionProposal(request)	发送交易提案
sendTransaction(request)	发送交易

更多详细的SDK接口实现请参考<https://github.com/hyperledger/fabric-sdk-node/tree/master/fabric-client/lib>。

5.4.2 SDK 应用开发

前面我们详细地介绍了如何使用CLI命令行去启动网络和操作链码。本节将介绍一个弹珠资产转移的例子。本例子的仓库地址为<https://github.com/IBM-Blockchain/marbles.git>。

1. 编写代码

marbles.go是此应用实例的智能合约实现链码，代码如下：

```
package main
import (
    ...
)
// 链码实现
type SimpleChaincode struct { }
// 实体对象定义，Marbles和Owners
// ----- Marbles 对象----- //
type Marble struct {
    ObjectType string    `json:"docType"` // 用于couchdb
    Id          string    `json:"id"`      // id
    Color       string    `json:"color"`   // marble颜色
    Size        int       `json:"size"`    // marble大小
    Owner       OwnerRelation `json:"owner"`   // 拥有者
}
// ----- Owners对象 ----- //
type Owner struct {
    ObjectType string `json:"docType"` // 用于couchdb
    Id string `json:"id"` // id
    Username string `json:"username"` // 用户名
    Company string `json:"company"` // 用户公司
}
// Marbles和持有者关系表，用于查询
type OwnerRelation struct {
    Id string `json:"id"` // id
    Username string `json:"username"` // 用户名
    Company string `json:"company"` // 公司
}
// Main方法
func main() {
    err := shim.Start(new(SimpleChaincode))
    if err != nil { fmt.Printf("Error starting Simple chaincode - %s", err) }
}
// Init方法
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("Marbles Is Starting Up")
    // 获取参数
    _, args := stub.GetFunctionAndParameters()
    var Aval int
```



```

var err error
if len(args) != 1 { return shim.Error("Incorrect number of arguments. Expecting 1") }
// 将numeric转为integer
Aval, err = strconv.Atoi(args[0])
if err != nil { return shim.Error("Expecting a numeric string argument to Init()") }
// 写入state marbles_ui
err = stub.PutState("marbles_ui", []byte("3.5.0"))
if err != nil { return shim.Error(err.Error()) }
// 启动一个测试
err = stub.PutState("selftest", []byte(strconv.Itoa(Aval)))
if err != nil {
    //测试失败
    return shim.Error(err.Error())
}
//测试通过
fmt.Println(" - ready for action")
return shim.Success(nil)
}
// Invoke方法
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    function, args := stub.GetFunctionAndParameters()
    fmt.Println(" ")
    fmt.Println("starting invoke, for - " + function)
    // 处理不同的方法调用
    if function == "init" {
        return t.Init(stub) // 初始化链码状态
    } else if function == "read" {
        return read(stub, args) // 形成readset
    } else if function == "write" {
        return write(stub, args) // 形成writeset
    } else if function == "delete_marble" {
        return delete_marble(stub, args) //从state中删除marbles
    } else if function == "init_marble" {
        return init_marble(stub, args) //创建一个新的marble
    } else if function == "set_owner" {
        return set_owner(stub, args) //更改marble的拥有者
    } else if function == "init_owner" {
        return init_owner(stub, args) //创建一个新的marble拥有者
    } else if function == "read_everything" {
        return read_everything(stub) //读取 (owners + marbles + companies)
    } else if function == "getHistory" {
        return getHistory(stub, args) //读取marble的历史信息
    } else if function == "getMarblesByRange" {
        return getMarblesByRange(stub, args) //读取marbles集合
    }
    // 出错
    fmt.Println("Received unknown invoke function name - " + function)
    return shim.Error("Received unknown invoke function name - '" + function + "'")
}

```

接下来这段代码展示了如何使用HFC客户端与Hyperledger区块链进行交互的过程。

```

enrollment.enroll = function (options, cb) {
    var chain = {};

```

```

var client = null;
try {
// Step 1 创建HFC客户端
client = new HFC();
    chain = client.newChain(options.channel_id);
}
catch (e) {
}
if (!options.uuid) { ... }
...
// Step 2 建立ECert kvs (Key Value Store)
HFC.newDefaultKeyValueStore({
// 在kvs目录中存储eCert
    path: path.join(os.homedir(), '.hfc-key-store/' + options.uuid) //store eCert in the kvs
    directory
}).then(function (store) {
    client.setStateStore(store);
// Step 3
    return getSubmitter(client, options);
}).then(function (submitter) {
// Step 4
    chain.addOrderer(new Orderer(options.orderer_url, {
        pem: options.orderer_tls_opts.pem,
        'ssl-target-name-override': options.orderer_tls_opts.common_name //can be null if
        cert matches hostname
    }));
// Step 5
    try {
        for (var i in options.peer_urls) {
            // 新建peer节点
            chain.addPeer(new Peer(options.peer_urls[i], {
                pem: options.peer_tls_opts.pem,
                'ssl-target-name-override': options.peer_tls_opts.common_name
            })); logger.debug('added peer', options.peer_urls[i]);
        }
    }
    catch (e) { }
    ...
// Step 6
// 打印日志
    logger.debug('[fcw] Successfully got enrollment ' + options.uuid);
    if (cb) cb(null, { chain: chain, submitter: submitter });
    return;
}).catch(
    function (err) { ... return; }
);
};

```

SDK调用过程如下。

- Step 1: 创建一个SDK实例。
- Step 2: 通过newDefaultKeyValueStore创建一个键值存储来存储登记证书。

- ❑ Step 3: 登记用户。这时候要用登记ID和等级密钥到CA获取认证, CA将会发布登记证书, SDK将其存在键值库中。若使用默认的键值库, 则会被存在本地文件系统中。
- ❑ Step 4: 成功登记后, 设置orderer URL。Orderer目前还不需要, 但是当调用链码时将会需要。“ssl-target-name-override”业务只有在你已经给证书签名的情况下需要。把这个字段设置为和你以前创建的PEM文件的“common name”一样。
- ❑ Step 5: 设置Peer的节点。这些暂时也不需要, 但需要设置好SDK的chain对象。
- ❑ Step 6: SDK已经完全配置好, 开始准备与区块链交互。

2. 应用运行

Marbles的应用交互流程如图5.5所示。

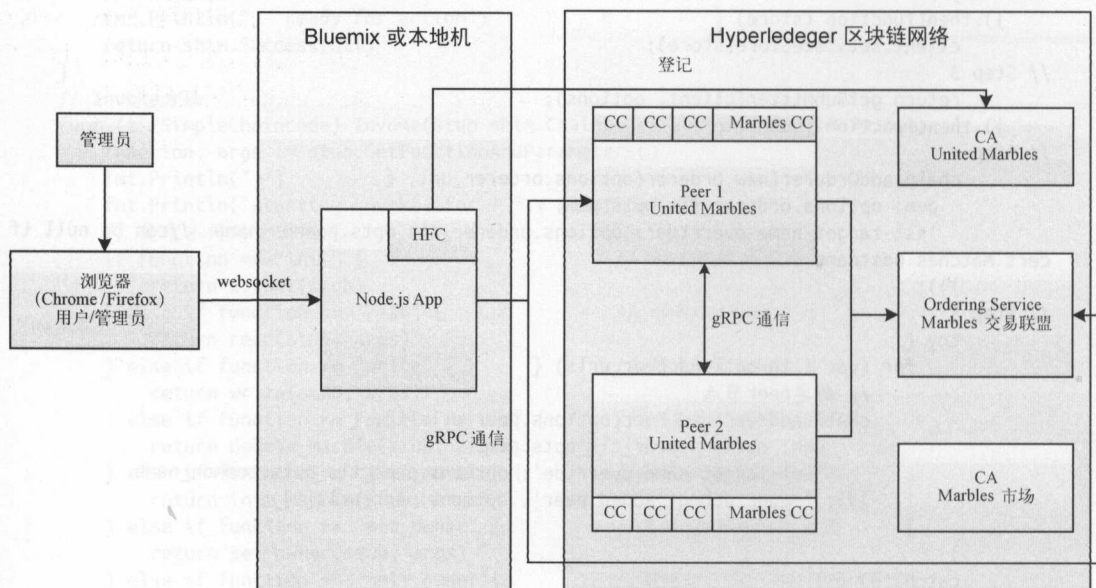


图5.5 应用交互流程

- (1) 浏览器与Node.js应用通过websocket服务进行通信。
- (2) Node.js应用与Hyperledger区块链网络间通过FabricNodeSDK (即HFC) 进行交互。
- (3) HFC与CA机构之间的通信基于HTTP协议。
- (4) HFC作为Hyperledger区块链网络中的客户端节点, 区块链网络中的节点互相之间基于gRPC协议进行通信。

应用运行步骤如下。

- ❑ Step 1: 参照实例1中的步骤配置网络和启动网络。
- ❑ Step 2: 配置JSON文件: /config/marbles1.json和/config/blockchain_creds1.json。

❑ Step 3: 安装和实例化链码，链码路径为/`chaincode/src/marbles`。

❑ Step 4: 启动应用。

```
npm install gulp -g
```

```
npm install
```

```
gulp
```

```
# 成功后会看到以下显示结果
```

```
----- Server Up - localhost:3000 -----
```

完成之后就可以在浏览器中输入`localhost:3000`进行访问了。

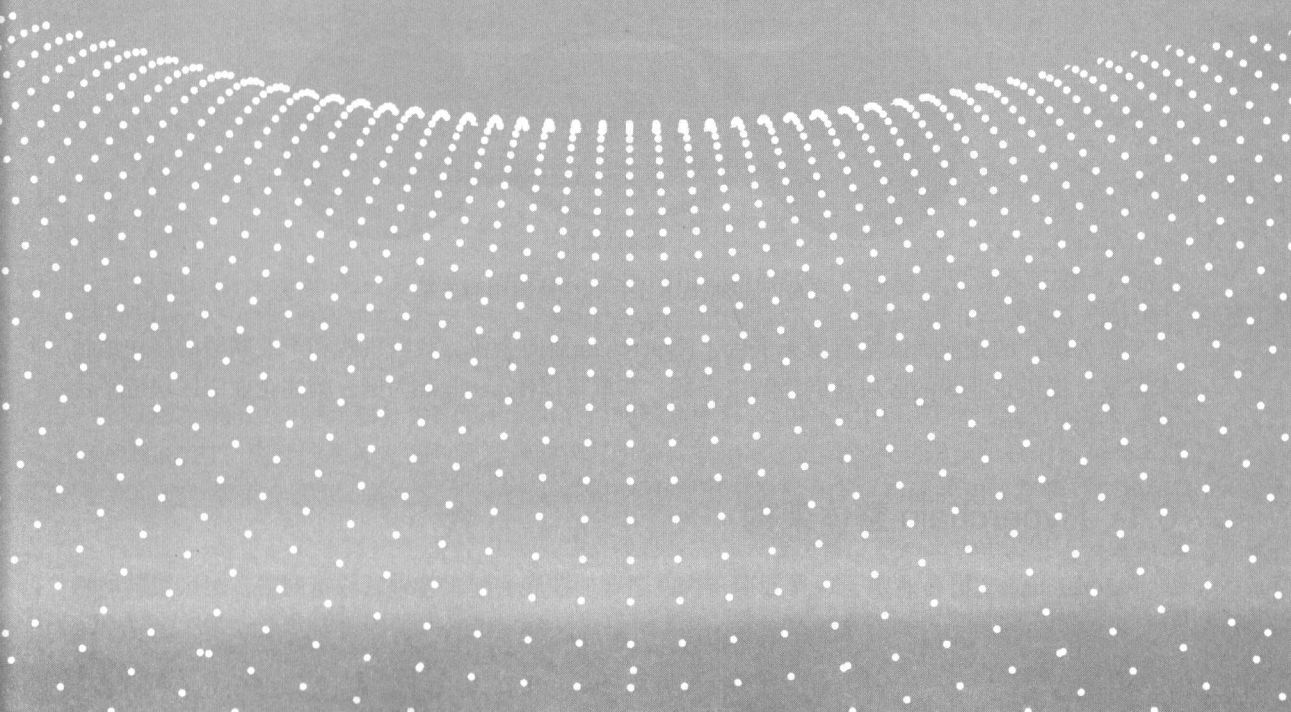
5.5 本章小结

本章主要介绍了如何在Hyperledger Fabric平台上开发区块链应用，首先讲述了Hyperledger Fabric的开发运行环境的搭建过程，然后讲解了链码开发和部署流程，最后介绍了CLI应用接口和SDK接口，并通过实例说明了如何基于这两种接口开发Hyperledger Fabric的区块链应用。

第三部分

企业级区块链平台 Hyperchain

- 第 6 章 企业级区块链平台核心原理剖析
- 第 7 章 Hyperchain 应用开发基础



企业级区块链平台核心原理剖析

企业级区块链（也称联盟链）主要针对大型公司、政府机构和产业联盟的区块链技术需求，提供企业级的区块链网络解决方案。联盟链的各个节点通常对应一个实体的机构组织，节点的加入和退出需要经过授权。各个机构组成利益相关的联盟，共同维护区块链网络的健康运转。

与私有链和公有链不同，企业级区块链更加着眼于区块链技术的实际落地，在区块链的性能速度和安全性、隐私性保护上有着更高的要求。除此之外，企业级区块链的研发往往直接和实际业务场景相关联，更加贴近行业痛点，为企业联盟提供一套更加完善的一体化区块链解决方案。图6.1展示了联盟链平台和区块链应用之间的相互促进关系。一方面，联盟链平台为实际行业应用研发、落地提供了底层技术支撑；另一方面，行业应用以及概念的验证落地也推动着联盟链平台的不断发展成熟。

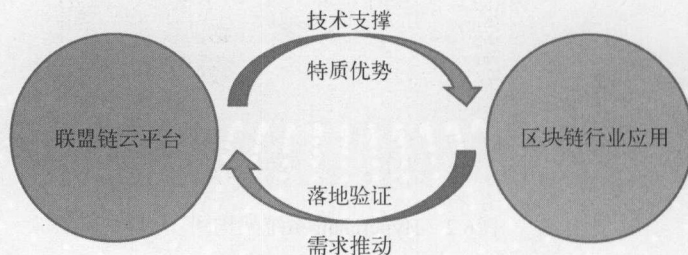


图6.1 联盟链云平台和行业应用的关系

作为国产的企业级区块链服务平台，Hyperchain面向企业、政府机构和产业联盟的区块链技术需求，提供企业级的区块链网络解决方案。本章将以Hyperchain为例，阐述企业级区块链平台设计的核心原理。

6.1 Hyperchain 整体架构

Hyperchain支持企业基于现有云平台快速部署、扩展和配置管理区块链网络，对区块链网络

的运行状态进行实时可视化监控，是符合ChinaLedger技术规范的国产区块链核心系统平台。Hyperchain具有验证节点授权机制、多级加密机制、共识机制、图灵完备的高性能智能合约执行引擎等核心特性，是一个功能完善、性能高效的联盟链基础技术平台。在面向企业和产业联盟需求的应用场景中，Hyperchain能够为资产数字化、数据存证、供应链金融、数字票据、支付清算等多中心应用提供优质的底层区块链支撑技术平台和便捷可靠的一体化解决方案。Hyperchain的整体系统架构如图6.2所示。



图6.2 Hyperchain系统架构图

首先，Hyperchain通过多种API与SDK对上层应用提供服务，为应用开发者屏蔽了区块链底层实现细节。目前提供了Restful、JSON-RPC等API，以及Java、JavaScript、Go等语言的SDK。这些外部编程接口通过内部的HTTP服务器同Hyperchain平台进行交互。

其次，图6.2的中间部分为Hyperchain平台的核心组件，包括基于PBFT改进的可靠高性能共识算法RBFT，高性能图灵完备的智能合约执行引擎，支持动态成员加入与退出、权限控制与多级加密的企业级安全模块。除此之外，Hyperchain还包含用于交易、区块存储的相关存储管理组件。

最后，为了企业级区块链平台的管理以及智能合约的开发方便，Hyperchain提供了一套可视化监管平台Hypervision。Hyperchain既支持直接裸机的安装部署，也可以使用Docker、Kubernetes

等集群管理工具进行资源管理方式的部署。平台可以部署在物理机以及OpenStack、Aliyun、AWS等主流的云平台上，适配各种服务器配置，适用各种主流操作系统。

本章接下来就以Hyperchain为例，阐述构成企业级区块链平台的核心技术模块，主要就共识算法、智能合约、账本、安全机制以及可视化监控平台的实现原理进行深入分析。

6.2 共识算法

共识算法是保证区块链平台各节点账本数据一致的关键，目前常见的分布式系统一致性算法包括PoW、PoS、Paxos、Raft、PBFT等。其中PoW依赖机器的计算能力获取账本的记账权，资源消耗较高且可监管性弱，每次交易共识的达成需要全网共同参与计算，因此不适合联盟链对监管以及性能的要求。PoS的主要思想是节点获得记账权的难度与其持有的权益数量成反比，相比PoW性能较好，但是依然存在可监管性弱的问题。Paxos和Raft是传统分布式系统的一致性成熟解决方案，此类型算法的性能高、消耗资源低，但是不具备对拜占庭节点的容错。PBFT算法同Paxos算法的处理流程类似，是一种许可投票、少数服从多数的共识机制。该算法具备容忍拜占庭错误的能力，且能够允许强监管节点的参与，算法性能较高，适合企业级平台的开发。目前主流的企业级区块链解决方案Fabric和Hyperchain都提供了PBFT的实现方案。然而原生PBFT算法在可靠性与灵活性方面不够完善，Hyperchain平台对可靠性与灵活性进行了增强，设计实现了PBFT的改进算法，即RBFT（Robust Byzantine Fault Tolerant）。

6.2.1 RBFT 概述

Hyperchain的共识模块采用可拔插的模块化设计，能够针对不同的业务场景需求选择配置不同的共识算法，目前支持PBFT的改进算法RBFT。Hyperchain通过优化PBFT的执行过程，增加主动恢复与动态节点增删等机制，极大地提高了传统PBFT的可靠性与性能。RBFT能够将交易的延时控制在300 ms，并且最高可以支持每秒上万笔的交易量，为区块链的商业应用提供了稳定高性能的算法保障。下面就RBFT的核心算法进行详细阐述。

6.2.2 RBFT 常规流程

RBFT的常规流程保证了区块链各节点以相同的顺序处理来自客户端的交易。RBFT同PBFT的容错能力相同，需要至少 $3f+1$ 个节点才能容忍 f 个拜占庭错误。图6.3中的示例为最少集群节点数，其 f 的值为1。图中的Primary为区块链节点中动态选举出来的主节点，负责对客户端消息的排序打包，Replica节点为备份节点，所有Replica节点与Primary节点执行交易的逻辑相同，Replica节点能够在Primary节点失效时参与新Primary节点的选举。

RBFT的共识保留了PBFT原有的三阶段处理流程（PrePrepare、Prepare、Commit），但是穿插增加了重要的交易验证环节。

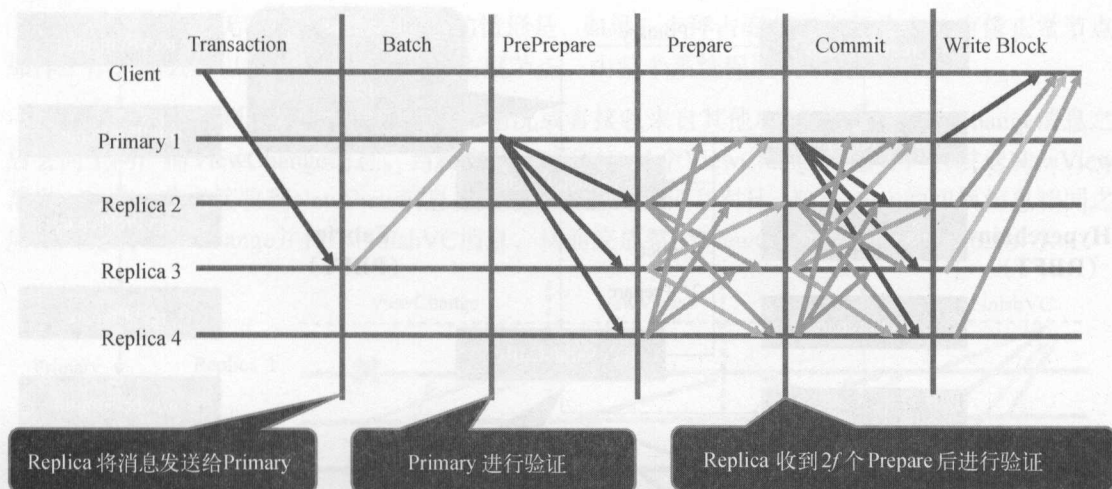


图6.3 RBFT常规共识流程

RBFT算法的常规共识流程如下所示。

- (1) Client将交易发送到区块链中的任意节点。
- (2) Replica节点接收到交易之后转发给Primary节点，Primary自身也能直接接收交易消息。
- (3) Primary会将收到的交易进行打包，生成batch进行验证，剔除其中的非法交易。
- (4) Primary将验证通过的batch构造PrePrepare消息广播给其他节点。
- (5) Replica接收来自Primary的PrePrepare消息之后构造Prepare消息发送给其他Replica节点，表明该节点接收到来自主节点的PrePrepare消息并认可主节点的batch排序。
- (6) Replica接收到 $2f$ 个节点的Prepare消息之后对batch的消息进行合法性验证，验证通过之后向其他节点广播Commit消息，表示自己同意了Primary节点的验证结果。
- (7) Replica节点接收到 $2f+1$ 个Commit之后执行batch中的交易并同主节点的执行结果进行验证，验证通过将会写入本地账本。

由以上的RBFT常规流程可以看出，RBFT将交易的验证流程穿插于共识算法的整个流程中，做到了对写入区块结果的共识。首先，Primary节点接收到交易之后首先进行验证，这保证了平台的算力不会被非法交易所消耗，使Replica节点能够高效地处理Primary节点的拜占庭失效。其次，Replica节点在接收到 $2f$ 个Prepare消息之后对Primary节点的验证结果进行验证，如果结果验证不通过则会触发ViewChange消息，这再一次保证了系统的安全性。图6.4是RBFT的共识流程与传统PBFT算法验证的具体流程对比图。

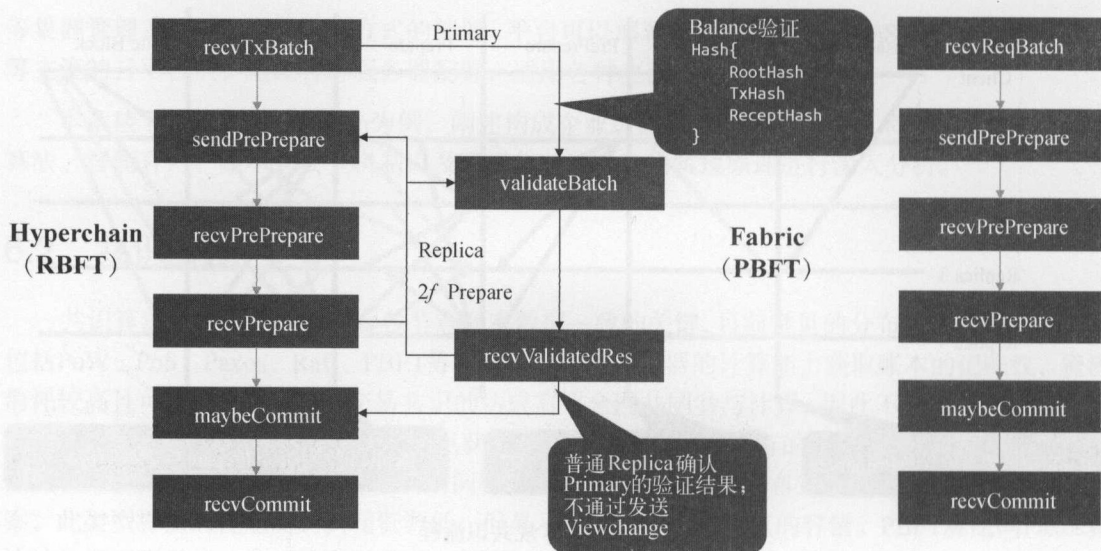


图6.4 RBFT流程与PBFT流程对比

6.2.3 RBFT 视图更换

在PBFT算法中，参与共识的节点可根据角色分为主节点（Primary）和从节点（Replica），从节点会将自己收到的交易转发给主节点，主节点最重要的功能就是将收到的所有交易按照一定策略打包成块，让所有节点参与共识验证。那么，一个很自然的问题就是，如果主节点发生宕机、系统错误或者被攻占（即成为拜占庭节点），其他从节点如何才能及时发现主节点的异常并选举产生新的Primary继续共识？这是保证BFT类算法稳定性必须要解决的问题。

在PBFT以及RBFT中都引入了视图（View）的概念，每次更换一个Primary节点同时切换视图，ViewChange（视图更换）机制是保证整个共识算法健壮性的关键。

目前能够检测到的主节点的拜占庭行为有3种情景：(1)节点停止工作，不再发送任何消息；(2)节点发送错误的消息，错误可能是消息内容不正确、包含恶意交易的消息等，需要注意的是，这里的消息类型不仅是batch，也有可能是用于ViewChange的功能性消息；(3)伪装正常节点，发送正确的消息。

对于情景(1)，可以由nullRequest机制保证，行为正确的主节点会在没有交易发生时向所有从节点发送nullRequest来说明这一情况的属实性，如果从节点在规定时间内没有收到主节点的nullRequest，则会引发ViewChange行为选举新的Primary。

对于情景(2)，从节点在接收主节点的消息时，通过验证机制检测，都会对内容进行相应的判断，如果发现主节点的交易包含不符合相应格式的交易或者恶意交易，即验证不通过的时候，会发起ViewChange选举新的Primary。

对于情景(3), 无需考虑, 一个极端的情形是, 如果一个拜占庭节点在行为上一直像正常节点那样工作, 那么可以认为它不是一个拜占庭节点, 由整个系统保证结果的正确性。

Replica节点检测到主节点有以上异常情况或者接收来自其他 $f+1$ 个节点的ViewChange消息之后会向全网广播ViewChange消息。当新主节点收到 $N-f$ 个ViewChange消息时, 会发送NewView消息。Replica节点接收到NewView消息之后进行消息的验证和对比, 验证View的切换信息相同之后正式更换ViewChange并打印FinishVC消息, 从而完成整个ViewChange流程, 如图6.5所示。

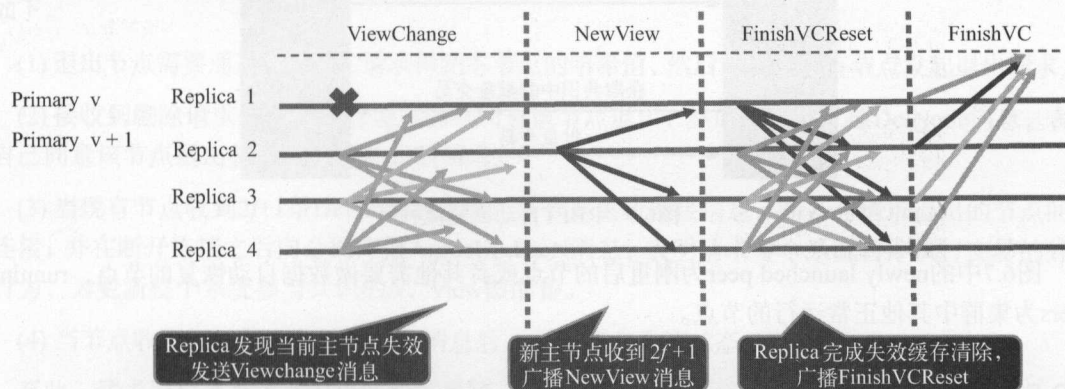


图6.5 RBFT ViewChange示意图

6.2.4 RBFT 自动恢复

区块链网络在运行过程中由于网络抖动、突然断电、磁盘故障等原因, 可能会导致部分节点的执行速度落后于大多数节点或者直接宕机。在这种场景下, 节点需要能够做到自动恢复并将账本同步到当前区块链的最新账本状态, 才能参与后续的交易执行。为了解决这类数据恢复工作, RBFT算法提供了一种动态数据自动恢复机制。

RBFT的自动恢复机制通过主动索取区块和正在共识的区块信息使自身节点的存储尽快和系统中的最新存储状态一致。自动恢复机制大大增强了整个区块链系统的可用性。RBFT为了恢复的方便, 对执行的数据设置checkpoint (检查点), checkpoint是通过全网共识的结果, 其格式如图6.6所示。这样就保证了每个节点上的checkpoint之前的数据都是一致的。除了checkpoint之外, 还有部分数据存储的是当前还未共识的本地执行进度。这样在恢复的过程中, 首先需要本节点的checkpoint点与区块链其他正在正常服务节点的checkpoint同步。其次, 需要恢复检查点之外的部分数据。图6.6为checkpoint的示意图, 左边为checkpoint部分, 右边为当前执行检查点之外的部分。图6.7所示是自动恢复机制的基本处理流程图。



图6.6 RBFT checkpoint示意图

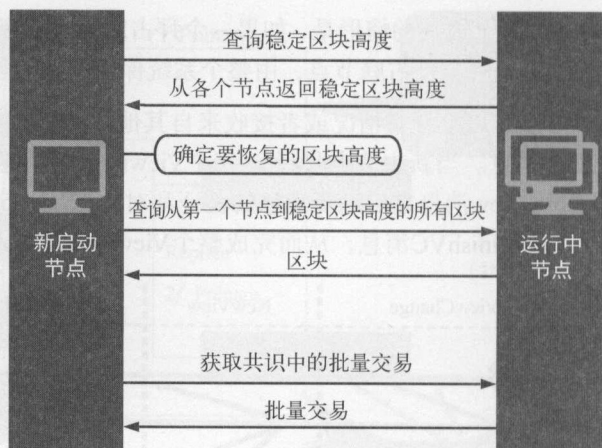


图6.7 RBFT自动恢复流程

图6.7中的newly launched peer为刚重启的节点或者其他需要做数据自动恢复的节点，running peers为集群中其他正常运行的节点。

6.2.5 RBFT 节点增删

在联盟链的场景下，由于联盟的扩展或者某些成员的退出，需要联盟链支持成员的动态进出服务，而传统的PBFT算法不支持节点的动态增删。RBFT为了能够更加方便地控制联盟成员的准入和准出，为PBFT添加了保持集群非停机的情况下动态增删节点的功能。如图6.8所示，RBFT为新节点加入算法处理流程。

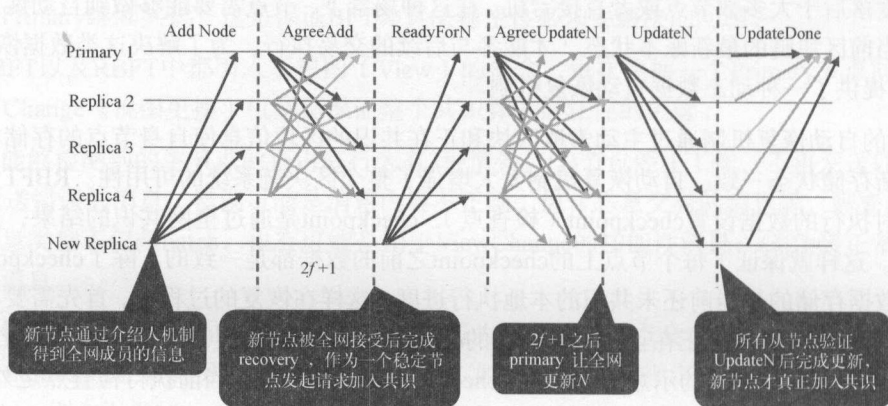


图6.8 RBFT节点增删

首先，新的节点需要得到证书颁发机构颁发的证书，然后向联盟中的所有节点发送请求。各个节点确认同意后 would 向联盟中的其他节点进行全网广播，当一个节点得到 $2f+1$ 个同意加入的回复

后会与新的节点建立连接。随后，当新的节点和 $N-f$ (N 为区块链联盟节点总数)个节点建立连接后就可以执行主动恢复算法，同步区块链联盟成员的最新状态。随后，新节点再向主节点请求加入常规共识流程。最后，主节点确认过新节点的请求后会定义在哪个块号后需要改变节点总数 N 来共识（确保新节点的加入不会影响到原有的共识，因为新节点的加入会导致全网共识 N 的改变，意味着 f 值可能改变）。

RBFT节点的动态删除和节点的动态增加流程类似，其主要处理函数如图6.9所示，其主要流程如下。

- (1) 退出节点需要通过调用RPC请求得到本节点的哈希值，然后向全网所有节点发起退出请求。
- (2) 接收到删除请求的节点的管理员确认同意该节点退出，然后向全网广播DelNode消息，表明自己同意该节点退出整个区块链共识的请求。
- (3) 当现有节点收到 $2f+1$ 条DelNode消息后，该节点更新连接信息，断开与请求退出的节点间的连接；并在断开连接之后向全网广播AgreeUpdateN消息，表明请求整个系统暂停执行交易的处理行为，为更新整个系统参与共识的 N ，view做准备。
- (4) 当节点收到 $2f+1$ 个AgreeUpdateN消息后，更新节点系统状态。

至此，请求退出节点正式退出区块链系统。

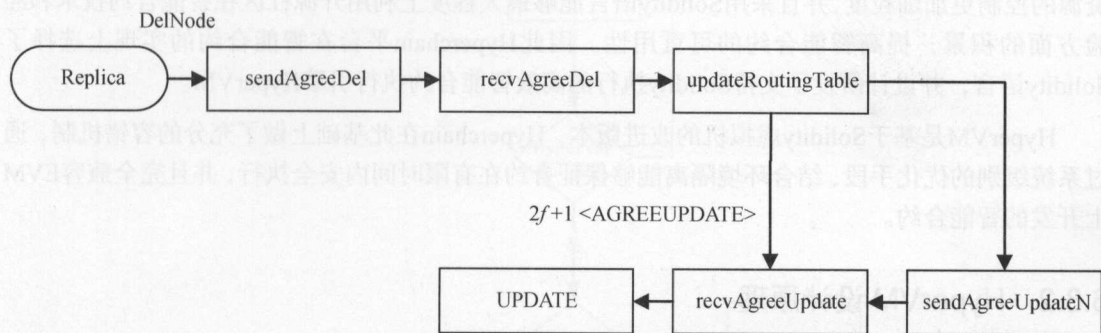


图6.9 动态节点退出函数调用

以上便是Hyperchain改进版的共识算法RBFT的主要算法流程。RBFT通过增加常规共识流程中的验证步骤，增加节点自动恢复机制，增加动态节点加入以及删除等功能，比传统PBFT算法更加稳定、灵活、高效，可以更好地满足企业级联盟链的生产环境需求。

6.3 智能合约

智能合约是部署在区块链上的一段可以自动执行的程序，广泛意义上的智能合约包含编程语言、编译器、虚拟机、事件、状态机、容错机制等。其中，对应用程序开发影响较大的是编程语言以及智能合约的执行引擎，即虚拟机。虚拟机作为沙箱被封装起来，整个执行环境都被完全隔

离。虚拟机内部执行的智能合约不能接触网络、文件系统或者系统中的其他线程等系统资源。合约之间只能进行有限调用。

目前智能合约的编写及其运行环境有3种典型实现范例：(1) IBM的Hyperledger Fabric项目用Docker作为智能合约的执行环境；(2) R3 Corda项目中的智能合约使用JVM作为合约的底层执行环境；(3) Ethereum项目中的智能合约采用Solidity进行编写，并使用内嵌型的Solidity虚拟机进行执行。

6.3.1 智能合约执行引擎

智能合约本质上就是一段程序，存在出错的可能性，甚至会引发严重问题或连锁反应。因此，智能合约的执行引擎的安全性对企业区块链的安全性来说至关重要。

Solidity是一种语法与JavaScript相似的高级语言，它专为智能合约的编写而设计。其编写十分简单，是一门图灵完备的语言，更重要的是它只能用来实现合约的逻辑功能，不提供任何访问系统资源的接口（例如打开文件、访问操作系统底层资源等），这在语言层面上就保证了用Solidity编写的智能合约能且只能运行在一个独立于操作系统的沙盒中，无法操纵任何系统资源。而Fabric基于Docker形式的虚拟机对语言并未进行特殊限制，因此安全性上不能做到完全保证。

除此之外，与Docker和JVM相比，Solidity语言及其智能合约执行引擎在程序体积上更小，对资源的控制更加细粒度，并且采用Solidity语言能够最大程度上利用开源社区在智能合约技术和经验方面的积累，提高智能合约的可重用性。因此Hyperchain平台在智能合约的实现上选择了Solidity语言，并设计研发了支持Solidity执行的高效智能合约执行引擎HyperVM。

HyperVM是基于Solidity虚拟机的改进版本，Hyperchain在此基础上做了充分的容错机制，通过系统级别的优化手段，结合环境隔离能够保证合约在有限时间内安全执行，并且完全兼容EVM上开发的智能合约。

6.3.2 HyperVM 设计原理

HyperVM的设计如图6.10所示，主要包括用于合约编译的编译器，用于代码执行优化的优化器，用于合约字节码执行的解释器，用于合约执行引擎安全性控制的安全模块，以及用于虚拟机和账本交互的状态管理模块。

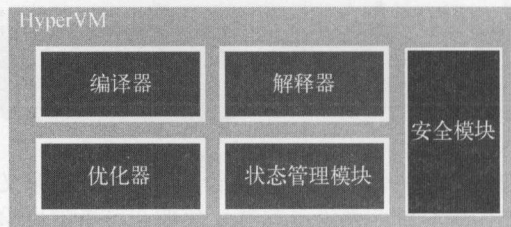


图6.10 HyperVM组件图

6.3.3 HyperVM 执行流程

图6.11是HyperVM执行交易的典型流程图，HyperVM执行一次交易之后会返回一个执行结果，系统将其保存在被称为交易回执的变量中，之后平台客户端可以根据本次的交易哈希进行交易结果的查询。

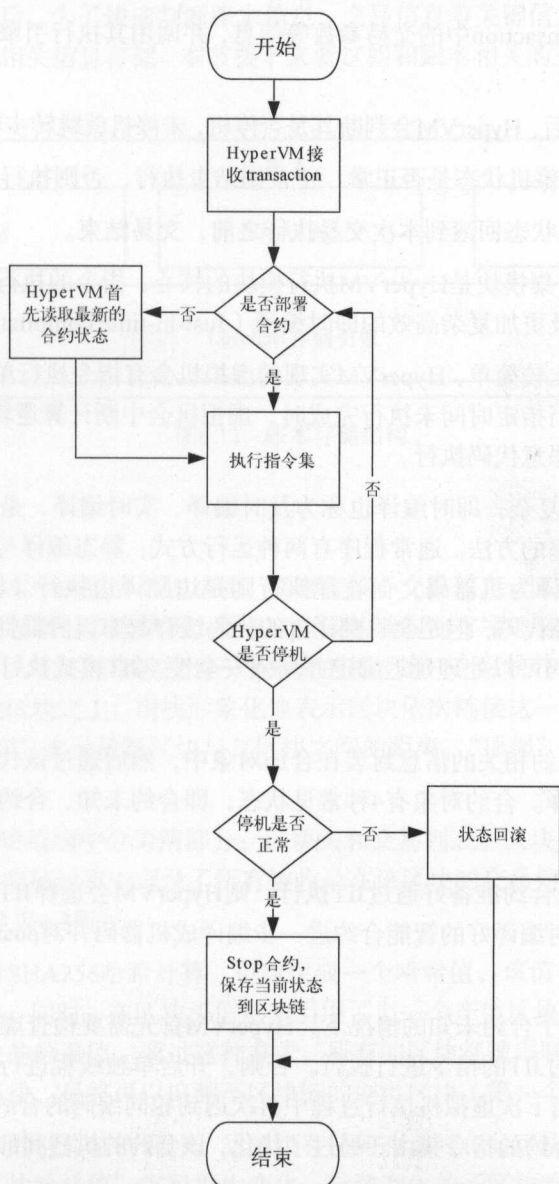


图6.11 HyperVM执行流程图

HyperVM的具体执行流程如下。

- (1) HyperVM接收到上层传递的transaction, 并进行初步的验证。
- (2) 判断transaction的类型, 如果是部署合约则执行步骤(3), 否则执行步骤(4)。
- (3) HyperVM新建一个合约账户来存储合约地址以及合约编译之后的代码。
- (4) HyperVM解析transaction中的交易参数等信息, 并调用其执行引擎执行相应的智能合约字节码。
- (5) 指令执行完成之后, HyperVM会判断其是否停机, 未停机就跳转步骤(2), 否则执行步骤(6)。
- (6) 判断HyperVM的停机状态是否正常, 正常则结束执行, 否则执行步骤(7)。
- (7) 进行Undo操作, 状态回滚到本次交易执行之前, 交易结束。

图6.11中的执行指令集模块是HyperVM执行模块的核心, 指令的执行模块有两种实现, 分别是基于字节码的执行以及更加复杂高效的即时编译(Just-in-time compilation, JIT)。

字节码执行的方式比较简单, HyperVM实现的虚拟机会指令执行单元。该指令执行单元会一直尝试执行指令集, 当指定时间未执行完成时, 虚拟机会中断计算逻辑, 返回超时错误信息, 以此防止智能合约中的恶意代码执行。

JIT方式的执行相对复杂, 即时编译也称为及时编译、实时编译, 是动态编译的一种形式, 是一种提高程序运行效率的方法。通常程序有两种运行方式: 静态编译与动态直译。静态编译的程序在执行前全部被翻译为机器码, 而直译执行则是边翻译边执行。即时编译器则混合了这二者, 一句一句地编译源代码, 但是会将翻译过的代码缓存起来以降低性能损耗。相对于静态编译代码, 即时编译的代码可以处理延迟绑定并增强安全性。JIT模式执行智能合约主要包含以下步骤。

(1) 将所有同智能合约相关的信息封装在合约对象中, 然后通过该代码的哈希值去查找该合约对象是否已经存储编译。合约对象有4种常见状态, 即合约未知、合约已编译、合约准备好通过JIT执行、合约错误。

(2) 如果合约状态是合约准备好通过JIT执行, 则HyperVM会选择JIT执行器来执行该合约。执行过程中虚拟机将会对编译好的智能合约进一步编译成机器码并对push、jump等指令进行深度优化。

(3) 如果合约状态处于合约未知的情况下, HyperVM首先需要检查虚拟机是否强制JIT执行, 如果是则顺序编译并通过JIT的指令进行执行。否则, 开启单独线程进行编译, 当前程序仍然通过普通的字节码编译。下次虚拟机执行过程中再次遇到相同编码的合约时, 虚拟机会直接选择经过优化的合约。这样合约的指令集由于经过了优化, 该合约的执行和部署的效率能够获得较大的提高。

6.4 账本数据存储机制

区块链本质上是一个分布式账本系统,因此区块链平台的账本体系设计至关重要。Hyperchain的账本设计主要包含3个部分:首先对客户的信息通过区块链这种链式结构进行存储,保证了客户交易的不可篡改以及可追溯性;其次,采用账户体系模型维护区块链系统的状态,即图6.12中的合约状态部分;最后,为了快速判断账本信息、交易信息等关键信息是否存在,账本采用了改进版的Merkle树进行相关信息存储。本节接下来就这些和账本相关的重要数据结构的设计进行详细分析。

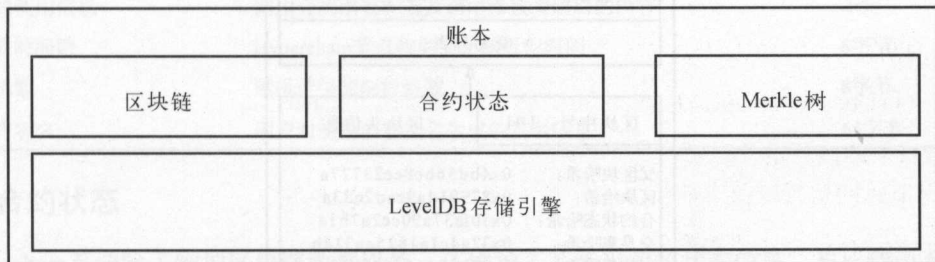


图6.12 账本存储结构

6.4.1 区块链

区块链是区块链账本中的重要数据结构,存储着核心交易信息。区块链是由包含交易信息的区块从后向前有序链接起来的数据结构。所有区块被从后向前有序地链接在这个链条里,每一个区块都指向其父区块。区块链经常被视为一个垂直的栈,第一个区块作为栈底的首区块,随后每个区块都被放置在其他区块之上。用栈形象化地表示区块依次链接这一概念后,我们便可以使用一些术语,例如,“高度”表示最新区块与首区块之间的距离,“顶部”或“顶端”表示最新添加的区块。

如图6.13所示,区块结构中分为两部分:区块头和交易列表。区块头中记录了一些固定大小的区块元数据信息,在交易列表中记录了所有被收录在该区块的交易信息。区块中的相应存储内容的具体定义如表6.1至表6.3所示。

对每个区块头进行SHA256哈希计算,可以生成一个哈希值,该值可以用作在区块链中唯一标识该区块的数字指纹。同时,在区块头信息中引用了上一个产生区块的哈希值,即在每一个区块中,都包含其父区块的哈希值。通过这种方式,所有的区块都被串联成一个垂直的链式结构,通过不断迭代访问父区块,最终可以追溯至区块链的创世区块(第一个区块)。

正是由于这种特殊的链式结构设计,父区块有任何改动时,父区块的哈希值也会发生变化,迫使子区块中的“父区块哈希值”字段发生变化,导致产生的子区块哈希值变化。Hyperchain节点之间每隔一个checkpoint会进行一次最新区块哈希的比较,如果本地维护的最新区块哈希值与

区块链网络维护的最新区块哈希值一致，则能确定本地维护的区块链信息是合法的，否则表示本地节点已经成为了一个“拜占庭节点”。



图6.13 区块链结构

表6.1 Hyperchain区块定义

字段名	描述	大小
区块头	区块元数据集	203字节
交易列表	收录在区块里的交易信息	可变

表6.2 Hyperchain区块头定义

字段名	描述	大小
版本信息	区块结构定义版本信息	3字节
父区块哈希	父区块哈希值	32字节
区块哈希	区块内容的哈希标识	32字节
区块号	区块高度	8字节
区块时间戳	主节点构造区块的近似时间	8字节
合约状态哈希	所有合约账户状态的哈希标识	32字节
交易集哈希	区块中收录的交易列表哈希标识	32字节
回执集哈希	执行交易产生的回执列表哈希标识	32字节
其他	区块执行时间戳，区块入链时间戳等	24字节

区块的交易列表中存储了被收录的交易数据，每条交易包含以下字段，如表6.3所示。

表6.3 交易结构定义

字 段 名	描 述	大 小
版本信息	交易结构定义版本信息	3字节
交易哈希值	根据交易内容生产的哈希标识	32字节
交易发起者地址	长度为40的16进制字符串，用于标识发起者	20字节
交易接收者地址	长度为40的16进制字符串，用于标识接收者	20字节
合约调用信息	调用合约函数标志及调用参数编码后的内容	不定
交易时间戳	Hyperchain节点收到交易的近似时间	8字节
随机数	随机产生的64位整数	8字节
用户签名	用户对交易内容签名生成的签名信息	65字节

6.4.2 合约状态

Hyperchain系统除了维护区块链数据以外，还维护了系统当前的状态信息。与比特币系统采用UTXO模型不同，Hyperchain采用了账户模型来表示系统状态。

当Hyperchain节点收到一笔“待执行”的交易后，会首先交由执行模块执行。执行交易结束后，会更改相关合约账户的状态，例如某用户A发起一笔交易调用已部署的合约B，使得合约B中的变量值 b 由0变为1，并持久化到合约状态中存储。

每一笔交易的执行，即意味着合约账户状态的一次转移，也代表着系统账本的一次状态转移。因此，Hyperchain也可以被认为是一个状态转移系统。

在Hyperchain账本中，会记录链上所有合约的状态信息。合约状态元数据共有以下几个字段，如表6.4所示。

表6.4 合约账户定义

字 段 名	描 述	大 小
合约地址	用于标识合约账户的唯一标识	20字节
合约存储空间哈希标识	利用Merkle树计算合约存储空间的所得的标识	32字节
合约代码哈希标识	合约可执行代码哈希产生的标识	32字节
创建者	创建该合约的账户地址	20字节
创建区块高度	合约被部署时的区块高度	8字节
合约状态	当前合约的可访问状态（正常或冻结）	1字节

除以上元数据以外，合约账户还有两个数据字段：可执行代码以及变量存储空间。可执行代码就是一段用字节数组编码的指令集，每一次合约的调用其实就是一次可执行代码的运行。合约

中定义的变量则会被存储在合约所属的存储空间中, 合约账户存储空间示意图如图6.14所示。

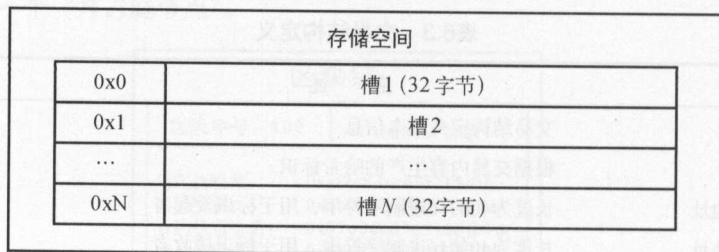


图6.14 合约账户存储空间示意图

存储空间与标准的存储结构类似, 在逻辑上是由一片地址连续的存储单元组成的(为了节省磁盘存储空间, 空的存储单元不被写入磁盘)。每一个存储单元称为一个槽, 大小为32字节。合约变量通过在合约编译阶段得到其在存储空间的索引地址, 内容存储在相应的槽中。

一个简易的合约状态数据示意图如图6.15所示。

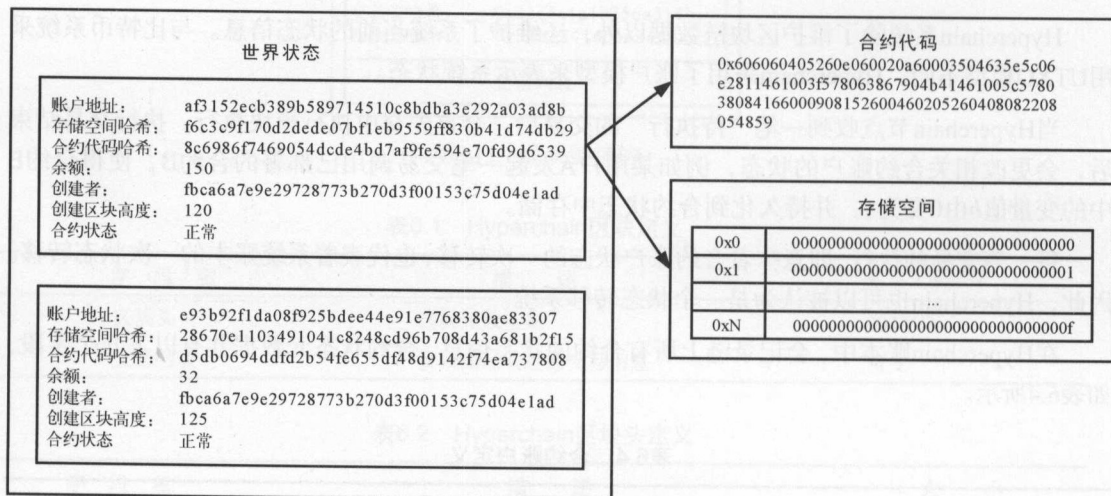


图6.15 合约状态数据示意图

6.4.3 Merkle 树

将区块中收录的交易依次处理之后, 合约账户从原先的状态转移至一个新的状态, 为了快速生成一个用于标识所有合约账户集新状态的哈希值, Hyperchain系统中引入了Merkle树进行哈希计算, 接下来先简明扼要地介绍一下Merkle树的结构和作用。

Merkle树是一种哈希二叉树, 它是一种用作快速归纳和校验大规模数据完整性的数据结构。这种二叉树包含加密哈希值, 在比特币网络中, Merkle树被用来归纳一个区块中的所有交易, 同

时生成整个交易集合的数字指纹,且提供了一种校验区块是否存在某交易的高效途径。但是传统的Merkle树性能较差,在面对高频海量数据时,计算的表现不能达到联盟链的需求。因此在Hyperchain中,设计了一种融合了Merkle树和哈希表两种数据结构各自优势的HyperMerkle树,大大提升了账本哈希计算的速率。

传统的Merkle树是自底向上构建的,如图6.16所示,从L1、L2、L3、L4这4个数据块开始构建Merkle树。首先对这4个数据块的数据哈希化,然后将哈希值存储至相应的叶子节点。这些叶子节点分别是Hash0-0、Hash0-1、Hash1-0和Hash1-1。

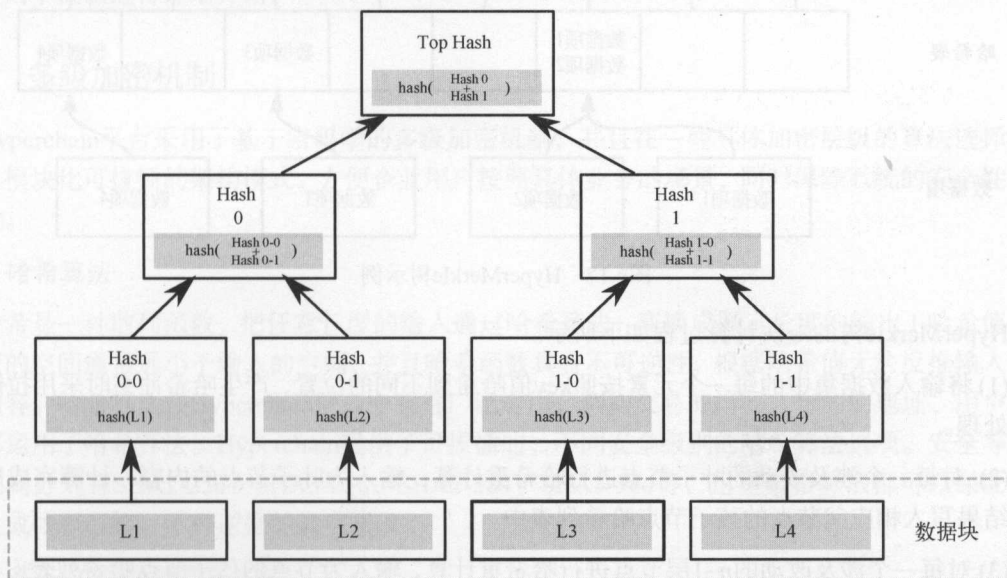


图6.16 传统Merkle树示例

完成最底层叶子节点的赋值之后,开始计算非叶子节点的值,计算方法为串联相邻叶子节点的哈希值,并以此为输入计算哈希,所得结果即为这对叶子节点父节点的哈希值。

继续类似的操作,直到只剩下顶部的一个节点,即Merkle根。根节点的哈希值即代表着这一批数据块的标识。

这种传统的Merkle树只适用于像比特币系统中对批量交易数据进行哈希的场景,而无法满足联盟链中快速计算账本哈希的需求。因此在Hyperchain中重新设计了结合哈希表特性的HyperMerkle树。

HyperMerkle树是一棵构建在哈希表上的多叉树,哈希表的每个存储单元均是HyperMerkle树的一个叶子节点,所有的叶子节点称为 n 层节点。将相邻若干个叶子节点归纳为一个父节点,生成的父节点集合称为 $n-1$ 层节点。递归上述操作直到只剩下顶部的一个节点即为HyperMerkle树的根节点。每个父节点维护着子节点哈希值列表。HyperMerkle树结构如图6.17所示。

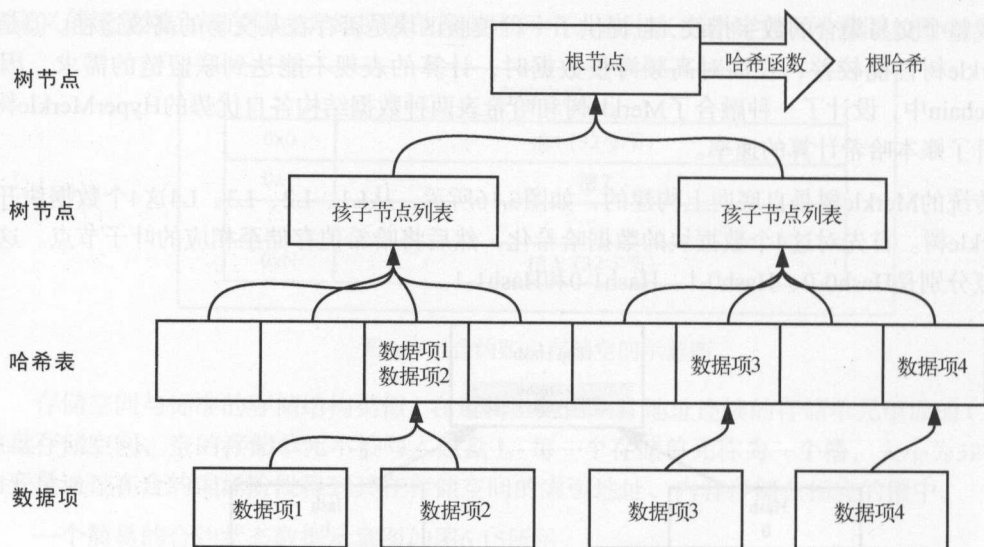


图6.17 HyperMerkle树示例

HyperMerkle树的一次计算过程如下所示。

(1) 将输入数据集中的每一个元素按照key值哈希到不同的位置, 产生哈希冲突时采用拉链法进行处理。

(2) 对每一个涉及改动的叶子节点进行哈希重计算, 输入为叶子节点的内容; 计算完成后将计算结果写入相应父节点的孩子节点哈希列表中。

(3) 对每一个涉及改动的 $n-1$ 层节点进行哈希重计算, 输入为节点的孩子节点哈希列表(本次计算未涉及的孩子节点的哈希值使用上次计算的值); 计算完成后将计算结果写入相应父节点的孩子节点哈希列表中。

(4) 重复步骤(3), 直至计算至1层节点。1层节点也称为根节点, 账本的当前哈希值用根节点哈希值表示。

(5) 将本次重计算的所有节点的内容持久化。

一棵HyperMerkle树维护一批数据, 且每次修改后只针对被修改的部分进行哈希重计算, 通过这种机制可以大幅提升计算效率。

HyperMerkle树在Hyperchain中具体进行两部分内容的哈希计算: 合约账户存储空间的哈希计算; 合约账户集的哈希计算。

对于每个合约账户, 存储空间的内容是HyperMerkle树的输入, 输出保存在合约账户的元数据中; 对于合约账户集, 每个合约的内容是HyperMerkle树的输入, 输出保存在区块中, 视作当前合约账户集状态的标识。

6.5 安全与隐私机制

企业级区块链平台也即联盟链，联盟链这个名词包含两层含义，首先它是区块链，其次它是有限成员联盟性质的。因此，在企业区块链安全性机制的设计上，既需要考虑传统区块链面对的各成员之间的信任问题，同时还要解决联盟成员的准入准出的安全管理机制。为此，Hyperchain平台提出了基于密码学的多级加密机制，在交易网络、交易双方以及交易实体等多个层面使用安全加密算法对用户信息进行了全方位加密，此外还提出了基于CA的权限控制机制，本节接下来将就这两个机制进行详细介绍。

6.5.1 多级加密机制

Hyperchain平台采用了基于密码学的多级加密机制，并且在一些具体加密层级的算法选择上实现了模块化可拔插的架构模式，方便企业用户按照具体业务的场景，同时保障系统的安全性和高效性。

1. 哈希算法

哈希是一种散列函数，把任意长度的输入通过哈希算法，变换成固定长度的输出（哈希值），哈希值的空间通常远小于输入的空间，并且哈希函数具有不可逆性，根据哈希值无法反推输入原文的内容。哈希算法在Hyperchain平台中有着广泛运用，例如交易的摘要、合约的地址、用户地址等都运用了哈希算法。Hyperchain提供了可拔插的、不同安全级别的哈希算法选项。安全等级由低到高分别有SHA2-256、SHA3-256、SHA2-384、SHA3-384等，这些哈希算法都可以保证为消息生成体积可控、不可逆推的数字指纹。

2. 基于ECDSA的交易签名

为了防止交易被篡改，Hyperchain平台采用了成熟的椭圆曲线数字签名算法（ECDSA）对交易进行签名。椭圆曲线密码体制的安全性基于椭圆曲线离散对数问题（ECDLP）的难解性。因此椭圆曲线密码系统的单位比特强度要远高于传统的离散对数系统。椭圆曲线密码系统（ECC）的主要优势是在某些情况下比其他方法使用更小的密钥，这带来的好处就是计算参数更小，密钥更短，运算速度更快，签名也更加短小。

在椭圆曲线加密标准与实现上，不同于Fabric直接采用Go官方库crypto包的ECDSA椭圆曲线数字签名算法，Hyperchain平台基于椭圆曲线secp256k1标准实现了数字签名算法。考虑到在数字签名以及签名验证过程中，涉及复杂且大量的计算，Hyperchain平台采取C语言封装的椭圆曲线加密标准，在签名和验证的性能上有更好的表现。

3. 基于ECDH的密钥协商

在网络通信过程中，使用会话密钥对传输的信息进行加密，可以防止黑客窃听机密消息进行欺诈等行为。然而，会话密钥的建立和网络中各用户之间的相互认证是需要认证密钥协商协议来

完成的。因此，区块链平台各个节点之间的密钥对需要特殊的协议进行协商。

Hyperchain通过实现椭圆曲线Diffie-Hellman (ECDH) 密钥协商协议，保证通信双方可以在不安全的公共媒体上创建共享的机密协议，而不必事先交换任何私有信息。任何截获交换的组织都能够复制公共参数和通信双方公钥，但是，无法从公开共享值生成共享机密协议。Hyperchain在通信双方协商出一个机密共享密钥后，再基于AES对称加密算法，将通信内容加密通过gRPC进行传输。这样可以保证从计算上破解传输内容的难度更高，从而保证消息传输的高安全性。

4. 同态加密

除此之外，Hyperchain还支持加法同态加密。通过智能合约的方式，保证了交易过程中或者智能合约中的数值在可以被交叉验证的同时，却对验证方不可见。这进一步保护了客户的数据隐私安全，更加符合经济活动的基本要求。

5. 国密算法

相比于其他区块链平台，Hyperchain在加密算法上有一个很大的优势，即Hyperchain完全支持国密算法的集成。目前Hyperchain集成了国密算法SM2和SM3。其中，SM2为基于椭圆曲线密码的公钥密码算法标准，包含数字签名、密钥交换和公钥加密，用于替换RSA/DiffieHellman/ECDSA/ECDH等国际算法；SM3为密码哈希算法，用于替代MD5/SHA-1/SHA-256等国际算法。

6.5.2 基于 CA 的权限控制

我们在6.5.1节中介绍了Hyperchain如何设计多级加密机制来保障企业区块链的交易安全、数据安全以及通信链路的安全。本节将介绍企业区块链中联盟成员的权限控制以及联盟成员的准入准出机制。Hyperchain的平台权限控制主要通过CA体系保证。Hyperchain的证书颁发体系如图6.18所示。

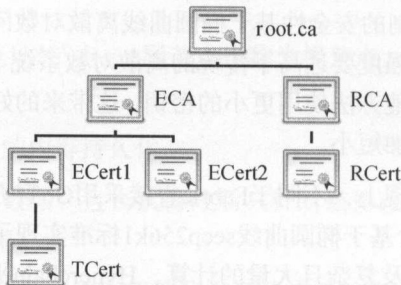


图6.18 Hyperchain证书颁发体系

首先简要介绍一下图中涉及的缩略词，ECA（Enrollment Certificate Authority）为准入证书颁发机构，该机构能够向下颁发准入证书。ECert（Enrollment Certificate）准入证书，由准入证书颁发机构颁发，持有ECert的节点才能够同Hyperchain链上服务交互。从图6.18还可以看出，

Hyperchain的ECert设计上有两种实现。持有ECert1的机构不仅拥有同Hyperchain链上服务交互的权限,还能够向下颁发TCert(Transaction Certificate)交易证书。交易证书用于实现伪匿名交易,主要在客户发起交易的时候使用,客户端会使用TCert相匹配的私钥对Transaction进行加密。TCert可以实现线上申请,由各个节点签发,每一条Transaction都可以用一个新的TCert进行签名,可以实现每条交易的相对匿名,但是可以由签发方审查。

RCA(Role Certificate Authority)为角色证书认证机构,该机构有权限颁发RCert(Role Certificate)。RCert主要是用于区分区块链节点中的验证节点和非验证节点,拥有RCert才被认为是区块链中的验证节点,参与区块链节点之间的共识。RCert和TCert一样只能作为身份证明的证书存在,不能向下颁发证书。

6.6 可视化监管平台

区块链监控管理平台是企业级区块链平台的关键组成部分,实时监控区块链平台健康状态,实时查询节点的区块生成状态,是保证企业区块链健康稳定的关键。Hyperchain设计实现了支持区块链平台监控、智能合约编写等功能于一体,方便部署于主流公有云、私有云或者物理机的企业级管控平台Hypervision。

6.6.1 Hypervision 概览

Hypervision的启动概览界面如图6.19所示。



图6.19 Hypervision概览

概览主要包括以下5个组成部分。

- **区块链监控**

该模块展示了最新区块号、区块链总体交易量、当前的TPS以及交易的延迟时间等信息。

- **服务器**

该模块是服务器状态的整体状态显示，显示了处于不同状态的服务器的数量，用户可以通过点击更多按钮跳转到具体服务器状态细节的界面。

- **交易数**

该部分展示当前24小时、当前7天、当前30天的交易数（横坐标为时间，纵坐标为交易数量）。

日报分24小时，展示每小时的交易总数；周报分7天，展示每天的交易总数；月报分30天，展示每天的交易总数。

- **容器**

该部分展示容器数量，不同状态的容器数量。当用户鼠标移动至容器某一饼图部分时，会显示该部分的容器数量。当用户点击该部分的“更多”时，会跳转至“区块链>>操作”界面。

- **智能合约**

该部分展示智能合约的具体信息；当用户鼠标移动至合约的进度条某一位置，会显示该合约数据。当用户点击该部分的“更多”时，会跳转至“智能合约”的详细管理界面。

6.6.2 Hypervision 区块链管理

Hypervision的区块链管理包括配置、操作、监控和浏览4个方面，相关操作界面如下所示。

- **配置界面**

该配置界面能够展示Hyperchain的配置信息，能够方便灵活地进行RBFT算法等关键组件的属性进行配置。

- **操作界面**

操作界面能够对容器进行相应管理，包括展示已添加的容器列表、添加并部署容器节点、清空容器以及查看Hyperchain节点日志。

- **监控界面**

该页面监控全局网络节点的实时状态，用户可随时查看全局区块链网络节点。当鼠标悬停在区块时间、交易数以及交易处理时间的柱状图时，会提示纵坐标信息，如图6.20所示。

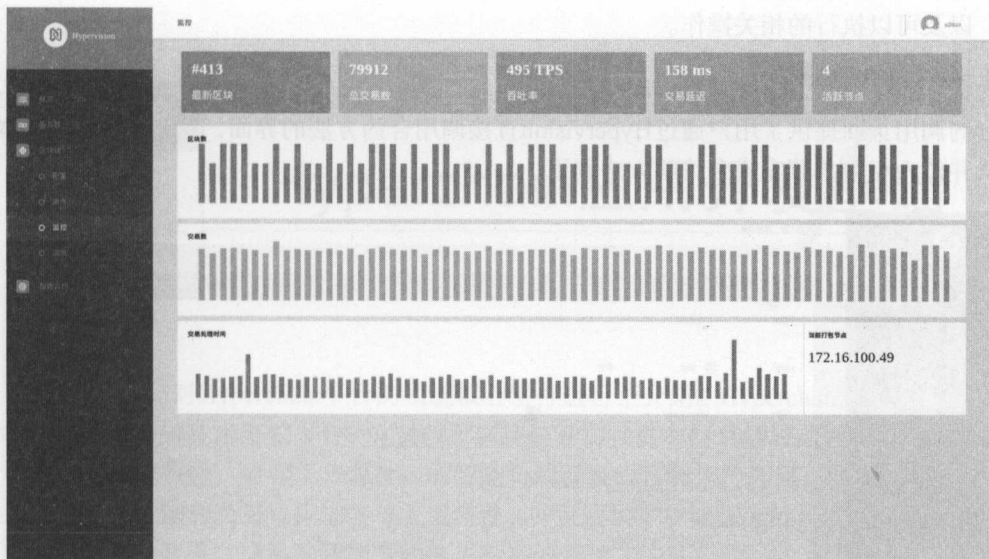


图6.20 区块链监控界面

- 浏览界面

该模块向用户展示了区块列表、区块详情、交易列表和交易详情页面。

- 区块列表界面

当用户点击浏览模块，默认进入区块列表页面时，该页面展示了区块的编号、哈希、写入时间、平均交易处理时间和交易数等信息。

6.6.3 Hypervision 合约管理

为了方便智能合约的编写、部署、管理，Hypervision为智能合约管理提供了相应的管理模块。合约管理模块主要包含以下功能的管理界面。

- 合约编辑界面

合约编辑的界面为用户提供了方便的、高亮显示的智能合约编写页面，除此之外，该界面还能够实现合约的编译，从而生成相应的字节码文件，如图6.21所示。

- 合约部署界面

合约部署页面提供了可视化的智能合约部署方式，根据合约部署方式的不同分为创建型和加载型两种。其中创建型合约用户可以直接编辑，而加载型则需要用户从本地上传译好的字节码文件。

- 合约查询界面

合约查询可以展示合约的具体信息，包括合约所在的项目名称、合约类型、合约状态、合约

地址，以及可以执行的相关操作。

• 合约调用界面

合约调用页面提供了用户通过Hypervision直接调用合约方法的界面，用户通过选择具体的合约方法并传入相关参数完成合约的方法调用。

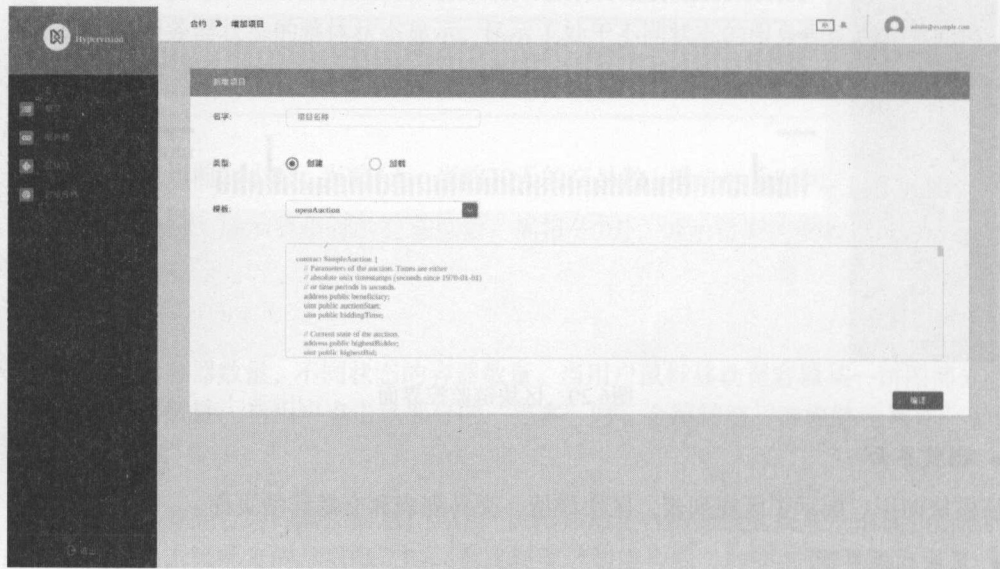


图6.21 编写合约界面

6.7 本章小结

本章以企业级区块链平台Hyperchain为例，介绍了构成企业级区块链平台的核心组件的实现原理。企业级区块链同公有链和私有链不同，它直接面对企业级应用的需求，对区块链系统的安全性、灵活性以及性能有着更加严格的要求。Hyperchain企业级区块链平台分别从以下几点出发，构建满足企业需求的区块链平台。首先，Hyperchain平台在优化传统PBFT的基础上设计实现了灵活、高效、稳定的共识算法RBFT，为企业级区块链平台提供了坚实的算法基础。其次，在智能合约的支持上选择了支持开源领域活跃的Solidity语言，并对其执行虚拟机进行了系统层面的优化，极大地提高了智能合约的执行速度。而且，在区块链账本的设计上选择了同比特币不同的账本体系，更加适合企业区块链的应用特性。再次，Hyperchain平台通过对交易、交易链路、应用开发包等多层面进行了加密处理，系统性地加强企业级区块链的安全等级。最后，为了企业级区块链的运维监控的便捷，Hyperchain还设计实现了支持系统监控、合约编写、合约编译等多功能的企业级区块链管控平台。

在第6章中，本书对Hyperchain平台的系统核心组件进行了介绍。Hyperchain是一个满足行业应用需求的联盟区块链技术基础平台，它吸收了区块链开源社区和研究领域的最前沿技术，集成了高性能的可靠共识算法，兼容于开源社区的智能合约开发语言和执行环境；同时，Hyperchain还强化了记账授权机制和交易数据加密等关键特性，并且提供了功能强大的可视化Web管理控制台，对区块链节点、账簿、交易和智能合约等进行高效管理。

对于这样一个企业级区块链核心平台，如何在Hyperchain上进行应用开发是开发者最关心的问题，也是本章介绍的重点。本章将先重点介绍平台所提供的基本功能与平台的配置部署，然后通过一个实际案例具体阐述如何使用Hyperchain平台进行区块链应用的开发。

7.1 平台功能

企业级区块链Hyperchain作为一种分布式一致性的账本系统，主要以交易处理的形式对外提供服务。Hyperchain中的交易分为两种：普通交易和智能合约交易。普通交易功能较为单一，只能提供相对基本的转账功能；而基于智能合约的交易则能够实现用户自定义的复杂逻辑，将交易逻辑封装于智能合约内部，因此更加灵活和安全。此外，Hyperchain还提供了许多有关区块链数据管理的相关功能，方便用户进行区块链数据的查询。

Hyperchain平台在应用接口方面对外提供了JSON-RPC、Java、Go等相关的软件开发服务。本章接下来将以JSON-RPC为例，分别就Hyperchain的主要服务交易调用、合约管理以及区块管理等方面进行详细说明。

7.1.1 JSON-RPC 格式

JSON-RPC是一种以JSON为数据传输格式的无状态且轻量级的远程过程调用（RPC）协议。Hyperchain底层平台对外以JSON-RPC的方式提供了应用访问区块链平台的操作入口，因此用户能够按照自身需求方便地使用任意语言进行接口调用，完成与区块链交互通信。每次通过JSON-RPC的方式发送一个请求对象至服务端，均代表一个RPC调用，而每个RPC调用的请求对象都需要包含下列成员字段。

- **jsonrpc**: 指定JSON-RPC协议版本的字符串, 如果是2.0版本, 则必须准确写为2.0 (Hyperchain使用的是JSON-RPC 2.0版本)。
- **method**: 表示所要调用方法名称的字符串。以RPC开头的方法名, 用英文句号 (U+002E or ASCII 46) 连接的为RPC内部预留的方法名及扩展名, 且不能在其他地方使用。
- **params**: 调用方法所需要的结构化参数值, 该成员参数可以被省略。
- **id**: 已建立客户端的唯一标识ID, 该值必须包含一个字符串、数值或NULL值。如果不包含, 该成员则被认定为是一次通知调用。该值一般不为NULL, 若为数值则应为整数。当发起一次RPC调用时, 服务端必须回复一个JSON对象作为响应, 响应对象包含下列成员。
 - **result**: 该成员在RPC调用成功时必须被包含, 当调用方法失败时必须不包含该成员。服务端中的被调用方法决定了该成员的值。
 - **error**: 该成员在RPC调用失败时必须被包含, 当没有错误发生时, 不包含该成员。若发生错误, 则该成员对象将包含**code**和**message**两个属性。
 - **id**: 该成员必须被包含。该成员值必须与请求对象中的id成员值一致。

7.1.2 交易调用

交易调用是Hyperchain应用主要使用的服务接口, 与交易调用相关的接口主要包括两类: 一类是用于查询交易的查询接口, 支持获取交易信息、查询交易详情、查询交易处理时间等查询操作; 另一类则是发起交易的调用, 主要是使用**tx_sendTansaction**方法完成调用, 该方法能够封装交易, 使得交易在区块链上执行, 并使得交易的结果以共识的方式存储在分布式各个Hyperchain节点上。表7.1展示了Hyperchain提供的所有交易相关接口, 其中**tx_sendTransaction**、**tx_getTransactionReceipt**、**tx_getTransactionByHash**这3个方法最为常用, 下面将以这3个接口为例, 讲述Hyperchain中交易相关接口的调用方式。

表7.1 交易服务接口概览表

RPC方法	功 能
tx_getTransactions	获取所有交易
tx_getDiscardTransactions	获取所有非法交易
tx_getTransactionByHash	根据交易哈希值查询交易详情
tx_getTransactionByBlockHashAndIndex	根据区块哈希值查询交易详情
tx_getTransactionByBlockNumberAndIndex	根据区块号查询交易详情
tx_sendTransaction	发送交易
tx_getTransactionsCount	查询链上所有交易量
tx_getTxAvgTimeByBlockNumber	查询指定区块中交易平均处理时间
tx_getTransactionReceipt	查询指定交易回执信息
tx_getBlockTransactionCountByHash	查询区块交易数量
tx_getSighHash	获取用于签名算法的哈希值

发送交易接口的方法tx_sendTransaction的详细说明如表7.2所示，该接口仅支持非智能合约的交易，比如用户之间的转账交易等普通交易。在调用tx_sendTransaction接口之前，需要先调用tx_getSighHash接口，获取用于客户端签名用的哈希，然后在客户端签名生成signature之后，再调用tx_sendTransaction接口方法，发送交易到区块链平台。

表7.2 发送交易接口

RPC方法	参 数	返 回 值
tx_sendTransaction	<pre> { from: <string>发起者地址 to: <string>接收者地址 value: <number>交易量值 timestamp: <number> 交易时间戳 signature: <string>签名 } </pre>	transactionHash: <string> 交易的哈希值，32字节的十六进制字符串

下面的Example 1展示了一个从账户0xb60e8dd61c5d32be8058bb8eb970870f07233155向另一个账户0xd46e8dd67c5d32be8058bb8eb970870f07244567转账2441406250的调用实例。如果该调用成功，则系统将会返回该交易的哈希值，后续客户端可以通过该哈希值查询该交易的详细信息。

Example 1（发送交易接口调用实例）：

```
// 请求
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "tx_sendTransaction",
  "params": [{
    "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
    "to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567",
    "value": 2441406250,
    "timestamp": 1477459062327000000,
    "signature": "your signature"
  }], "id": 71}'
// 返回结果
{
  "id": 71,
  "jsonrpc": "2.0",
  "result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
}
```

表7.3为根据哈希值查询交易信息的接口的详细描述，上述描述的发送接口发送交易成功之后，会返回一个表示本次交易的哈希，后续通过该哈希值可以查询该交易的具体信息。

表7.3 根据哈希查询交易接口表

RPC方法	参 数	返 回 值
tx_getTransactionByHash	transactionHash <string> 交易的哈希值, 32字节的十六进制字符串	<TransactionResult>TransactionResult 对象

TransactionResult 对象:

```
{
  hash: <string>交易的哈希值, 32字节的十六进制字符串
  blockNumber: <number> 交易所在的区块高度
  blockHash: <string> 交易所在区块哈希值
  txIndex: <number> 交易在区块中的交易列表的位置
  from: <string> 交易发送方的地址, 20字节的十六进制字符串
  to: <string>交易接收方的地址, 20字节的十六进制字符串
  amount: <number> 交易量
  timestamp: <number>交易发生时间 (单位ns)
  executeTime: <string> 交易的处理时间 (单位ms)
  invalid: <boolean> 交易是否不合法
  invalidMsg: <string> 交易的不合法信息
}
```

下面的Example 2展示了一个按照交易哈希查询交易信息的例子, 从交易的返回结果来看, 查询交易能够查询该交易的哈希值、交易存储的区块号以及交易的具体信息、交易执行时间等交易相关详细信息。

Example 2 (根据交易哈希查询交易实例):

```
// 请求
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "tx_getTransactionByHash",
  "params": ["0x658406ea4edf92f4b9d1589c3ea84d75c07f4179908e899703eae0e3ea54caa2"],
  "id": 71
}'
// 返回结果
{
  "jsonrpc": "2.0",
  "id": 71,
  "result": {
    "hash": "0x658406ea4edf92f4b9d1589c3ea84d75c07f4179908e899703eae0e3ea54caa2",
    "blockNumber": "0x1",
    "blockHash": "0x9e330e8890df02d22a7ade73b5060db6651658b676dc9b30e54537853e39c81d",
    "txIndex": "0x9",
    "from": "0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd",
    "to": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "amount": "0x1",
    "timestamp": 1477459062327000000,
    "executeTime": "0x3ee",
```



```

    "invalid": false,
    "invalidMsg": ""
  }
}

```

根据交易哈希查询交易的方法返回的交易结果为一个TransactionResult对象,它只是包含了交易详细信息,但是并未包含交易执行结果信息。交易的成功与否以及相应的返回结果需要通过交易回执信息接口方法tx_getTransactionReceipt进行查询,表7.4为tx_getTransactionReceipt调用的详细描述。

表7.4 根据交易哈希返回交易回执信息接口

RPC方法	参 数	返 回 值
tx_getTransactionReceipt	transactionHash: <string>交易哈希	<Receipt>

<Receipt>对象:

```

{
  txHash: <string> 交易哈希
  postState: <string>交易状态
  contractAddress: <string>合约地址
  ret: <string>执行的结果
}

```

Example 3 (根据交易哈希查询交易回执信息实例):

```

// 请求
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "tx_getTransactionReceipt",
  "params": ["0xb60e8dd61c5d32be8058bb8eb970870f07233155"],
  "id": 71
}'
// 返回结果
{
  "id": 71,
  "jsonrpc": "2.0",
  "result": {
    "txHash": "0x9e330e8890df02d22a7ade73b5060db6651658b676dc9b
30e54537853e39c81d ",
    "postState": '1',
    "contractAddress": '0xe04d296d2460cfb8472af2c5fd05b5a214109c
25688d3704aed5484f'
    "ret": "0x06060405260e060020a60003504633ad14af3811460305780
63569c5f6d146056578063d09de08a14606d575b6002565b34600257600
0805460043563fffffffff8216016024350163fffffffff19909116179055
5b005b3460025760005463fffffffff166060908152602090f35b3460025
760546000805463fffffffff19811663fffffffff9091166001011790555"
  }
}

```

7.1.3 合约管理

智能合约是区块链系统的核心组成部分，通过智能合约，应用开发者能够自定义更加复杂灵活的链上资产处理逻辑。Hyperchain平台为合约管理提供了一套相关的API，包括合约的编译、部署、调用等。Hyperchain提供的智能合约接口列表如表7.5所示。

表7.5 智能合约服务接口概览表

RPC方法	功 能
contract_compileContract	编译合约
contract_deployContract	部署合约
contract_invokeContract	调用合约
contract_getCode	获取合约编码
contract_getContractCountByAddr	获取合约数量
contract_encryptoMessage	获取同态余额以及转账金额
contract_checkHmValue	获取同态交易的验证结果
contract_maintainContract	维护合约（升级、冻结、解冻）
contract_getStatus	查询合约状态
contract_getCreator	查询合约部署者
contract_getCreateTime	查询合约部署时间
contract_getDeployedList	查询指定账户已部署合约的列表

接下来就合约管理接口的几个关键函数编译、部署和调用进行详细介绍。表7.6所示为智能合约编译的调用接口的详细描述，编译智能合约接口只需要将本地写好的智能合约的源码以参数的形式上传即可。至于合约如何编写的相关内容将在7.3.1节中介绍。compileCode为接口返回的编译结果，从描述中可以看出合约编译返回了ABI、bin以及types几个字段。其中ABI为Application Binary Interface（应用程序二进制接口）为合约中具体调用的接口以及参数信息；bin为合约的编译之后二进制字节码文件，若源码中有多个合约，则bin为顶层合约的字节码；types中则保存了编译的相关合约的名字。

表7.6 编译合约源码

RPC方法	参 数	返 回 值
contract_compileContract	<string>合约源码	<compileCode>

说明 合约实际上是在本地编译，务必保证本地已经安装solidity环境，否则无法编译。

<compileCode>对象:

```
{
  abi: <Array> 合约源码对应的abi数组
  bin: <Array> 合约编译而成的字节码
```

```
types: <Array> 对应合约的名称
}
```

Example 4:

```
// 请求
{
  "jsonrpc": "2.0",
  "method": "contract_compileContract",
  "params": ["contract Accumulator{ uint sum = 0; function increment(){ sum = sum + 1; }
    function getSum() returns(uint){ return sum; }}"],
  "id": 1
}

// 返回结果
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "abi": [{"{"constant":false,"inputs":[],"name":"getSum","outputs":
[{"name":"","type":"uint256"}],"payable":false,"type":"function"},{"constant\
: false,"inputs":[],"name":"increment","outputs":[],"payable":false,"type":"funct
ion"}]}],
    "bin": ["0x60606040526000600060005055604a8060186000396000f3606060405260e060020a600035
0463569c5f6d81146026578063d09de08a146037575b6002565b346002576000546060908152602090f35b3460025
76048600080546001019055565b00"],
    "types": ["Accumulator"]
  }
}
```

在合约编译完成之后，下一步则是将合约部署到链上，部署到区块链要求合约能够通过共识的方式部署到区块链的各个节点上。表7.7所示为合约部署接口方法的详细参数信息，其中from指合约部署者自己在区块链的账户地址。值得注意的是，合约部署也是一个交易，同sendTransaction接口类似，也是返回该交易的哈希值。如果希望进一步确认合约部署的结果，则需要通过该哈希值查询交易的详细内容，交易内容包含了合约部署成功与否的信息以及该合约的部署地址，合约地址则用于唯一定位以及在合约调用时使用。

表7.7 部署合约

RPC方法	参 数	返 回 值
contract_deployContract	<pre>{ from: <string>合约部署者地址 timestamp: <number> 交易时间戳（单位ns） payload: <string> 合约编码contract_complieContract 方法返回的bin signature: <string>交易签名 }</pre>	<pre>transactionHash: <string> 交 易的哈希值，32字节的十六进 制字符串</pre>

表7.10 根据区块号查询区块详细信息

RPC方法	参 数	返 回 值
block_getBlockByNumber	blockNumber: <blockNumber>区块号	<blockResult>

blockNumber可以是十进制整数或者十六进制字符串,也可以是latest字符串,表示最新的区块。

<blockResult>对象:

```
{
  version: <string>平台版本号
  number: <string>区块的高度
  hash: <string>区块的哈希值, 32字节的十六进制字符串
  parentHash: <string>父区块哈希值, 32字节的十六进制字符串
  writeTime: <number>区块的生成时间 (单位ns)
  avgTime: <number>当前区块中交易的平均处理时间 (单位ms)
  txCounts: <number>当前区块中打包的交易数量
  merkleRoot: <string> Merkle树的根哈希
  transactions: [<TransactionResult>] 区块中的交易列表
}
```

Example 7 (根据区块号查询区块信息实例):

```
// 请求
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "block_getBlockByNumber",
  "params": ["0x3"],
  "id": 1
}'
// 返回结果
{
  "jsonrpc": "2.0",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718
    b1a88fead34c914",
    "parentHash": "0x2b709670922de0dda68926f96cffbe48c980c4325d
    416dab62b4be27fd73cee9",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef2
    3bd4d8fc85204cb00ab6c",
    "transactions": [
      {
        "version": "1.0",
        "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5
        f98ed67179b59169ae",
```

```
bc723718bb1a88fead34c914",
```

区间的所有区块接口

```
block getPlainBlocks
```

from必须小于等于to, 否则会返回error。

`<plainBlockResult>`对象:

{

Example 8 (区间区块查询接口实例):

```
// 请求
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method":
  "block_getPlainBlocks",
  "params": [{"from":2,"to":3}],
  "id": 1
}'

// 返回结果
{
  "jsonrpc": "2.0",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.0",
      "number": "0x3",
      "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb
1a88fead34c914",
      "parentHash": "0x2b709670922de0dda68926f96cffbe48c980c4325d41
6dab62b4be27fd73cee9",
      "writeTime": 1481778653997475900,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23
bd4d8fc85204cb00ab6c"
    },
    {
      "version": "1.0",
      "number": "0x2",
      "hash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62
b4be27fd73cee9",
      "parentHash": "0xe287c62aae77462aa772bd68da9f1a1ba21a0d044e2c
c47f742409c20643e50c",
      "writeTime": 1481778642328872960,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23
bd4d8fc85204cb00ab6c"
    }
  ]
}
```

7.2 平台部署

Hyperchain区块链平台使用基于PBFT的改进算法RBFT作为平台的共识算法。该算法要求Hyperchain集群至少需要4台机器才能够正常启动。Hyperchain平台部署机器的配置需求如表7.12所示。

表7.12 Hyperchain基本配置需求表

硬件及网络设施	配置需求
处理器	Intel Xeon CPU E5-26xx v3 2GHz, 4核及其以上
内存	8GB及其以上
硬盘	1TB HDD
带宽	1000Mbit/s

7.2.1 Hyperchain 配置

在正式部署Hyperchain系统之前, 需要对其相关参数进行配置。解压Hyperchain安装包hyperchain.tar.gz之后, 修改其主要配置文件。

- **peerconfig.json**: 配置节点IP以及端口号

将4个节点的IP地址以及gRPC通信端口(默认8001)进行设置, 确保所有的端口均为可用。

- **pbft.yaml**: 配置共识算法

修改相应的节点数目Nodes(默认值为4), 修改batch size大小(默认为500), 表示500条transaction 将会被打包成一个区块; 修改timeout:batch大小(默认值为100ms), 即当交易并发数量较小, 未达到batch size上限时, 节点每隔100ms打包一个区块。

- **global.yaml**: 全局配置

请在了解了每个配置项的意义后, 根据需求修改。如无特殊需求, 可以直接使用默认配置。

- **LICENSE**: Hyperchain授权证书

由Hyperchain公司颁发, 用于合法授权使用, 有一定的使用期限, 如果得到了更加高级的授权证书, 可以将默认的LICENSE文件覆盖。

其他具体配置信息可以根据企业要求、硬件环境等信息进行具体的选择配置。

7.2.2 Hyperchain 部署

Hyperchain的部署分为两种情况: 初次部署和动态新增节点。

对于初次部署的情况, 首先将Hyperchain的二进制包分发到初始化的几个节点上, 并按照7.2.1节的方式进行配置即可。

对于动态新增节点而言, 还需要额外对准入证书以及peerconfig.json文件进行相应的配置。证书的配置文件为caconfig.toml, 其相关配置如下所示:

```

# 入链节点都需要以下配置
[ecert]
# eca的路径, 用于验证ecert
ca = "config/cert/eca.ca"
# ecert的路径
cert = "config/cert/ecert.cert"
# ecert的私钥路径
priv = "config/cert/ecert.priv"
# 若该节点为非验证节点, 则可以为空
[rcert]
# rca的路径, 用于验证rcert
ca = "config/cert/rca.ca"
# rcert的路径
cert = "config/cert/rcert.cert"
# rcert的私钥路径
priv = "config/cert/rcert.priv"
# 若所有节点都需要, 否则无法进行网络通信
[tlscert]
# tlscas的路径, 用于验证tlscert
ca = "config/cert/tlscert/tlscas.ca"
# tlscert的路径
cert = "config/cert/tlscert/tls_peer1.cert"
# tlscert私钥的路径
priv = "config/cert/tlscert/tls_peer1.priv"
serverhostoverride = "hyperchain.cn"
[check]
# 用于配置该节点是否开启rcert验证开关
checkercert = false
# 用于配置该节点是否开启tcert验证开关
checktcert = false

```

除此之外, 还需要对peerconfig.json文件进行修改, 确保新增节点配置文件正确, 一个新增节点的peerconfig.json文件应该是如下结构。

```

{
  "self": {
    "is_reconnect": false,
    "is_origin": false,
    "is_vp": true,
    "node_id": 5,
    "grpc_port": 8005,
    "local_ip": "127.0.0.1",
    "jsonrpc_port": 8085,
    "restful_port": 9005,
    "introducer_ip": "127.0.0.1",
    "introducer_port": 8001,
    "introducer_id": 1
  },
  "maxpeernode": 4,
  "nodes": [
    {
      "id": 1,
      "address": "127.0.0.1",
      "port": 8001,

```

```

    "rpc_port":8081
  },
  {
    "id":2,
    "address":"127.0.0.1",
    "port":8002,
    "rpc_port":8082
  },
  {
    "id":3,
    "address":"127.0.0.1",
    "port":8003,
    "rpc_port":8083
  },
  {
    "id":4,
    "address":"127.0.0.1",
    "port":8004,
    "rpc_port":8084
  }
]
}

```

本配置文件中需要注意的地方有以下几点。

- ❑ `is_origin` 字段表示该节点是否为创世节点，动态新增的节点均不是创世节点，因此设置为 `false`;
- ❑ `is_vp` 表示新增节点的角色为共识验证节点（VP）或非共识验证节点（NVP）;
- ❑ `introducer` 为介绍人节点的相关信息，一般选择创世节点为介绍人;
- ❑ `maxpeernodes` 填充新节点未加入前区块链网络的节点数;
- ❑ `nodes` 只需要填创世节点的信息即可。

7.2.3 Hyperchain 运行

Hyperchain 平台的运行方式比较简便，对于初次启动的 Hyperchain 平台而言，分别进入 4 台 Hyperchain 节点的 hyperchain 目录下运行 `./start.sh`，启动相应的 Hyperchain 服务，其启动成功的标志如图 7.1 所示。

```

[NOTIC] 16:51:44.595 GRPCPeerManager.go:113
[NOTIC] 16:51:44.595 GRPCPeerManager.go:114 All NODES WERE CONNECTED
[NOTIC] 16:51:44.595 GRPCPeerManager.go:115
[CRITI] 16:51:44.596 pbftprotocol.go:676 ===== Replica 1 finished negotiating view=0 / N=4
[CRITI] 16:51:44.614 pbftprotocol.go:698 ===== Replica 1 finished recovery, height: 0

```

图 7.1 Hyperchain 启动日志

Hyperchain 启动时，日志可能会显示某些异常情况，下列是一些启动异常情况及其处理方案。

• 通信异常

当单个节点出现图 7.2 中的错误时，说明 Hyperchain 集群中有一个节点宕机或者网络通信出现问

题,这时可以通过restart.sh脚本重启该节点,如图7.2所示,可以通过IP知道具体出现问题的节点。

```
2016/12/20 19:51:56 grpc: addrConn.resetTransport failed to create client transport: connection error: desc = "transport: dial tcp 127.0.0.1:8002: getsockopt: connection refused"; Reconnecting to [{"127.0.0.1:8002"} <nil>]
[ERROR] 19:51:56.531 peer.go:68 cannot establish a connection
[ERROR] 19:51:56.531 gRPCPeerManager.go:137 Node: 127.0.0.1 : 8002 can not connect!
[ERROR] 19:51:56.531 gRPCPeerManager.go:137 Node: 127.0.0.1 : 8002 can not connect!
[ERROR] 19:51:56.531 gRPCPeerManager.go:137 Node: 127.0.0.1 : 8002 can not connect!
[ERROR] 19:51:56.531 gRPCPeerManager.go:137 Node: 127.0.0.1 : 8002 can not connect!
```

图7.2 Hyperchain通信异常

当多个节点都出现以上问题时,需要停止发送交易,先关闭所有节点(stop.sh),在所有节点上运行start.sh重启。

• ViewChange异常

如图7.3所示,当在所有节点上看到这条消息,且消息都是来自相同的节点(例如节点2)时,说明节点2发生了异常情况,触发了ViewChange,这时候不用做额外处理,当节点2发送10次ViewChange都不成功时,会自动触发recovery,最终达成一致。

```
Replica 1 already has a view change message for view 3 from replica 2
```

图7.3 Hyperchain ViewChange异常

当在所有节点都看到类似消息而且不断出现时,先关闭所有节点(stop.sh),再用start.sh一一启动。

• Ignore duplicate异常

如果偶尔出现如图7.4所示的信息,是正常情况,可能是某一个节点的CPU负荷过高或网络通信不畅造成的。

```
Replica 2 ignoring prepare for view=2/seqNo=10: not in-wv, in view 1, low water mark 20
```

图7.4 Hyperchain Ignore duplicate异常

如果一直出现,可能是一个节点处理速度过慢使处理落后了,可以等节点自动发现后恢复(需要等待一段时间,落后50个块会触发恢复),也可以通过将落后的节点重启来恢复。

7.3 第一个 Hyperchain 应用

前面介绍了Hyperchain平台对外提供的相关服务,本节将通过一个具体案例,分析如何使用Hyperchain平台构建区块链应用。假设我们有一个模拟银行的案例,该案例要求能够模拟银行内部的存款、取款以及储户之间的转账业务。对于这种核心的金融业务,我们需要将用户资产相关数据的操作处理都通过区块链进行操作,并记录在区块链上。本案例实现了一个智能合约对用户资产进行管理,智能合约内部实现存款、取款和转账逻辑。

7.3.1 编写智能合约

Hyperchain平台目前支持使用Solidity语言进行智能合约的开发。Solidity是一种类似JavaScript语言的智能合约编写语言，在Solidity中，合约由一组函数以及数据定义组成，通过数据定义实现对用户资产的抽象描述；通过合约函数实现对用户数字资产操作规则的制定。具体请参考Solidity语法规范。

Solidity智能合约的每个合约以contract关键字修饰。这和Java语言中的class概念类似，在contract代码内部可以定义该contract的数据和操作器数据的相关函数。本例中声明了银行相关的属性：银行名（bankName）、银行编号（bankNum）以及银行状态字段（isInvalid）等。除此之外，合约中还存储了一个Address字段修饰的合约创建者地址，记录合约创建者可以实现合约调用的权限控制。合约代码如下：

```
contract SimulateBank{
    address owner;
    bytes32 bankName;
    uint bankNum;
    bool isInvalid;
    mapping(address => uint) public accounts;
    function SimulateBank(bytes32 _bankName, uint _bankNum, bool _isInvalid) {
        bankName = _bankName;
        bankNum = _bankNum;
        isInvalid = _isInvalid;
        owner = msg.sender;
    }
    function issue(address addr, uint number) return (bool) {
        if(msg.sender == owner) {
            accounts[addr] = accounts[addr] + number;
            return true;
        }
        return false;
    }
    function transfer(address addr1, address addr2, uint amount) return (bool) {
        if(accounts[addr1] >= amount) {
            accounts[addr1] = accounts[addr1] - amount;
            accounts[addr2] = accounts[addr2] + amount;
            return true;
        }
        return false;
    }
    function getAccountBalance(address addr) return(uint) {
        return accounts[addr];
    }
}
```

上述代码中使用了一个map结构维护用户及其资产的映射关系，issue函数实现了用户存款的业务，transfer实现了用户之间的资产转账的业务，而getAccountBalance函数实现了用户余额的查询功能。至此，模拟银行合约的编写结束，接下来继续阐述如何对该合约进行编译、部署和调用。

7.3.2 部署与合约调用

智能合约的部署和调用都应该通过区块链平台的共识算法使得合约能够部署到区块链上,并且合约资产状态的改变应该能够实现多节点同步。Hyperchain对外提供的Java SDK能够方便地进行智能合约的调用,下面是利用Hyperchain的Java SDK进行SimulateBank的issue方法调用的例子。

```
String contractSourceCode = "SimulateBank source code";
String ownerAddr = "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914"
//1. 编译合约
HyperchainAPI api = new HyperchainAPI(
"http://localhost:8081", HttpProvider.INTERNET, false);
CompileReturn result = api.CompileContract(contractSourceCode, 1);
//2. 部署合约
SingleValueReturn deployRs = api.deployContract(
ownerAddr, result.getBin().get(0), null, "password123", 1);
String contractAddr = api.getTransactionReceipt(
deployRs.toString(), 1).getContractAddress();
//3. 调用合约
FuncParamReal addr = new FuncParamReal(
"address", "0x23acc3e13d8124fe799d55d7d2af06223148dc7bbc7723718bb1a88fead34c914");
FuncParamReal number = new FuncParamReal("uint", "10000");
String input = FuncParamReal.encodeFunction("issue", addr, number);
api.invokeContract(ownerAddr, contractAddr, input, null, "password123", 1);
```

如上代码所示,首先需要实例化一个HyperchainAPI的实例,接着使用该实例进行合约的编译、部署和调用。合约编译之后只能返回一个交易哈希,需要通过该哈希去查询交易的结果,如上述代码部署合约中拿到的交易回执里的合约地址。合约地址是调用合约的一个必选项。此外,在进行方法调用之前,首先要将需调用的方法及其参数进行编码,最后通过invokeContract方法进行具体的合约调用。

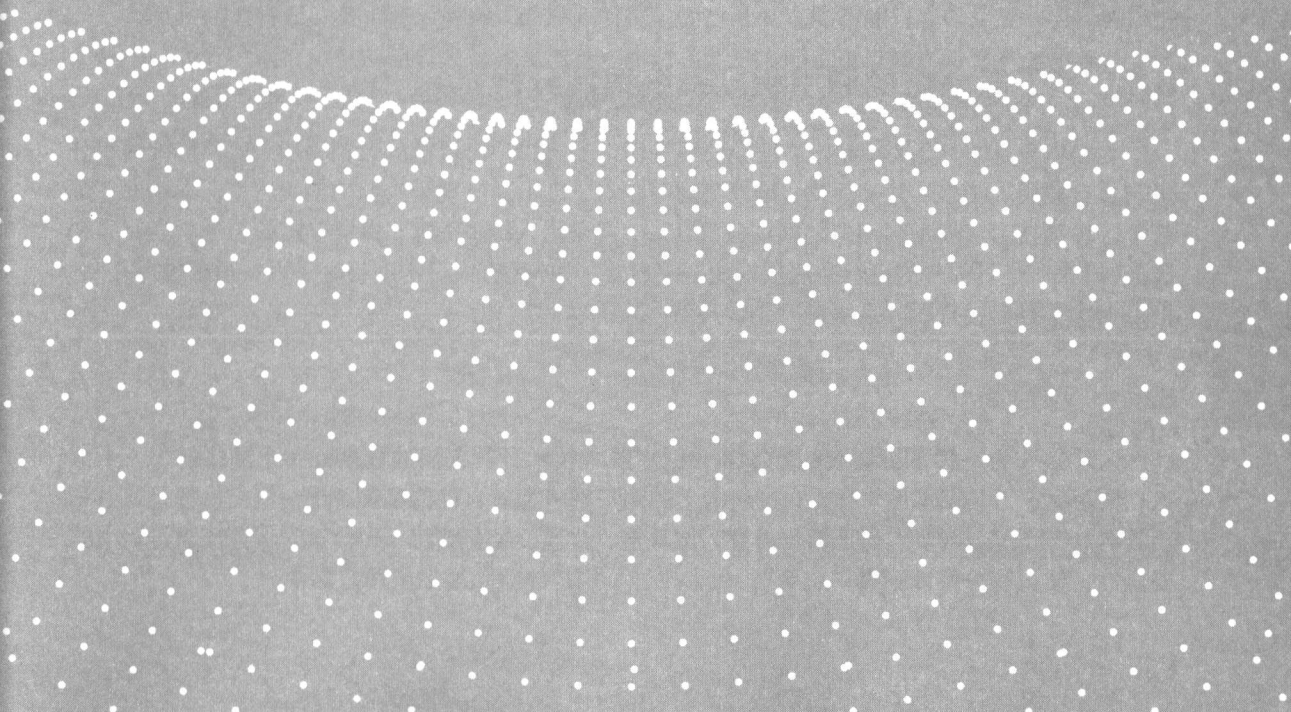
7.4 本章小结

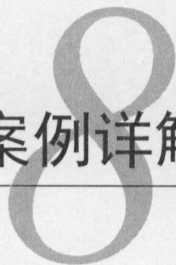
本章主要介绍了Hyperchain区块链上应用开发的相关内容。首先,本章从交易调用、合约管理以及区块查询几个方面介绍了Hyperchain平台对外提供的主要接口;其次,从Hyperchain集群的配置、部署和运行等方面介绍了如何搭建一个可运行的企业级区块链系统Hyperchain;最后,以模拟银行为例介绍了如何在Hyperchain平台上进行智能合约应用的开发。

第四部分

区块链应用案例

- 第 8 章 以太坊应用实战案例详解
- 第 9 章 企业级区块链应用实战案例详解





区块链技术作为当下有潜力触发颠覆性革命浪潮的核心技术，在金融领域的应用将有可能改变常规的交易流程和记录保存方式，从而大幅降低交易成本，提升效率。由于区块链安全、透明及不可篡改的特性，金融体系间的信任模式不再依赖中介，很多业务都将“去中心化”，实现实时数字化的交易。以太坊是知名的图灵完备的开源区块链平台，基于以太坊智能合约可以实现所有可计算的逻辑操作。众多区块链创新应用和初创公司的创新产品基于以太坊平台开发，本书前面的章节已经对以太坊的核心原理和开发实践进行了详细梳理，本章将更加贴近实战，介绍两个基于以太坊的区块链实际应用项目案例：通用积分系统和电子优惠券系统。其中，通用积分系统案例直接在以太坊底层平台上开发DApp应用，通过以太坊提供的web3.js接口，直接调用智能合约的方法，发送交易或读取数据，并在网页上展示给用户；而电子优惠券系统，整体架构是一个使用SSM框架的Java Web项目，在应用层引入区块链技术，自主封装与以太坊平台交互的接口工具，使用以太坊平台和数据库互相配合存储数据，并开发智能合约，将实际的后台业务逻辑作为交易记录到区块链上。

8.1 基于以太坊的通用积分系统案例分析

奖励积分是银行、大型超市、证券公司等用以提高用户忠诚度的营销手段，这种传统的积分机制具有使用限制多、兑换烦琐、难以流通等缺点，已不适应现今人们的消费习惯。本积分系统基于区块链技术实现不同用户之间积分的转让，并且引入线下商家，提供丰富的积分兑换奖品和服务。以太坊是目前最为流行的底层区块链平台之一，已经有大量的项目基于以太坊来进行开发。把以太坊平台与银行积分系统进行结合具有一定的实际意义。

8.1.1 项目简介

区块链作为一种不可篡改的分布式数据库账本技术，其存储的数据分布于网络的每一个节点，从而决定了其安全性。每一个区块链上的用户都将拥有自己的私钥，每一笔交易都是通过私钥签名的，经过全网节点认证后方可存入区块链，并且一经存储将不得修改，保证流通过程中的安全性，使得积分设计不再“鸡肋”，大大改善了用户体验，增加了用户黏性。

本系统的核心业务为银行积分的流通，简要流程为：银行可以向本行内的客户发行积分，客户可以将自己账户内的积分转让给其他客户或者商户，同时可以使用积分购买积分商城中的商品。商户可以向积分商城发布商品，每售出一件商品都可以获得相应的积分，商户可以向银行发起积分清算，把积分兑换成货币。系统的整体流程图如图8.1所示。

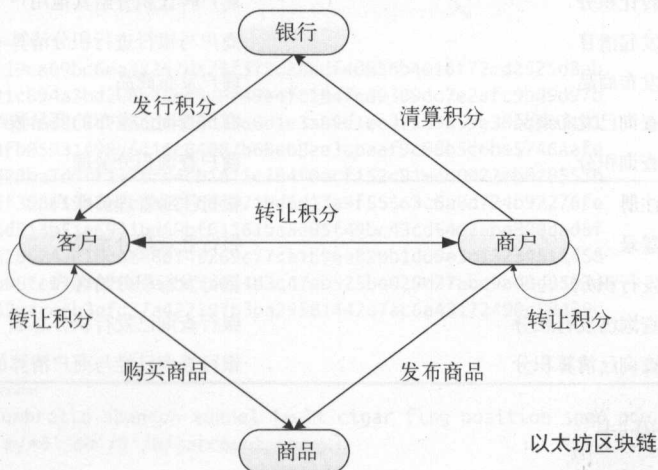


图8.1 积分系统流程图

8.1.2 系统功能分析

本系统主要涉及三类用户：客户、商户和银行。银行可以直接和商户进行交互，银行可以进行积分的发行，商户可以向银行发起积分清算。商户和客户之间也可以直接进行积分的流通，客户-客户、商户-商户、商户-客户两两之间都可以进行积分的转让，同时客户可以购买商户的商品，对应额度的积分会从客户账户流入商户账户。不同的用户都可以进行常见的查询操作。各个用户的具体功能如表8.1所示。

表8.1 通用积分系统需求要点

	需求要点	备 注
客户	注册	客户注册账户
	登录	客户登录积分系统
	转让积分	客户转让积分给其他用户（客户或商户）
	兑换商品	客户使用积分兑换商品
	查询已购买商品	客户查询已购买的商品数组
	查询积分	客户查询积分余额

(续)

	需求要点	备 注
商户	注册	商户注册账户
	登录	商户登录积分系统
	转让积分	商户转让积分给其他用户（客户或商户）
	发起清算	商户与银行进行积分清算
	发布商品	商户发布商品
	查询已发布商品	商户查询已发布的商品数组
	查询积分	商户查询积分余额
银行	注册	银行注册管理员账户
	登录	银行登录积分系统
	发行积分	银行发行积分给客户
	查询已发行积分	银行查询已发行积分总额
	查询已清算积分	银行查询已经与商户清算的积分总额

8.1.3 系统总体设计

本案例的总体设计主要包括方案选型和总体架构设计。方案选型包括以太坊客户端的选型、开发框架的选型和以太坊接口的选型；总体架构设计主要是底层区块链平台与上层业务之间的设计。良好的系统总体设计能为后续的智能合约设计和系统实现提供保障。

1. 方案选型

方案选型主要包括以太坊客户端、开发框架和接口类型。

(1) 以太坊客户端

在目前开发去中心化应用（DApp）中，TestRPC和geth这两种以太坊客户端使用较为普遍，本案例可以同时运行部署在TestRPC和geth中。但是在测试开发中，推荐TestRPC。TestRPC是基于Node.js开发的以太坊客户端，整个区块链的数据驻留在内存，发送给TestRPC的交易会被马上处理而不需要等待挖矿时间。TestRPC可以在启动时创建一堆存有资金的测试账户，它的运行速度也更快，因此更适合开发和测试。打开TestRPC的命令行界面如下：

```
→ ~ testrpc
Secp256k1 bindings are not compiled. Pure JS implementation will be used.
EthereumJS TestRPC v3.0.2
```

Available Accounts

=====

```
(0) 0x89af9f619b0e5394b545cecf38ced700abab10fd
(1) 0x13ae921ad067fd7ab50357442c33472b937529f7
(2) 0x2029ff200482fdbf9d3c1a20aff25b42fcc2b6c6
(3) 0x41b760eb63e1d008a2ad8ff7200f274c4c593872
```

```
(4) 0xb88f3ef66403279c3e507167c777c202703bf17
(5) 0xc41c5d74b4810d730080f4173529e2db8a9cf5aa
(6) 0x597ba89a34e3f3493a6f088bf87d598857d62207
(7) 0xbdb44c059820e2e3c2d8d221e312d39df6813a32
(8) 0xe59b4470d9845cf432f36242c731beb6530888d6
(9) 0xe8162329102720a910860742142b25651dfd5e17
```

Private Keys

```
=====
```

```
(0) 271341ed1ff19ca09bc6ea32257da73f372c28adf40856b4016f72cd2525d3ab
(1) 4cc0a47df621c094a3bd208e78be9def49e4fc1047ed9309dc7e2afc9bd9d97b
(2) 5ceffd1e45fe546d8c8d7e86a0438d3368d1e3a89d1ee37159659e30bd5e3101
(3) 628cbdef808fb05931498b6416c84087b60eb8ee3abaa9c08b5c6be5746aafe
(4) a97db67a68d80ba7d0ff13fbfd4cb1a12c284f0acf152c91e4d0022eb028553b
(5) 59fc5632606f308659a2efea60f68c7719d6d17a9f55563c6a0d724b92276fe
(6) 3e24ccde811d513b51e6941bd59bf01161ba5d05f49bc43cd5462abe800dad5f
(7) f66edd09d75285243e10cae880140269c77ca1b9ee820b1d09e3c13234511458
(8) a5b11d8daab082e07a1070a5aabc7a0c483c47eb525be029d27abc9ad0d8503e
(9) c93a6529b762e4cc6b3efc37a422797b3ba2958144267ac6a43172490c50409c
```

HD Wallet

```
=====
```

```
Mnemonic:      umbrella abandon tunnel fault cigar flag position snap now moment follow matrix
Base HD Path:  m/44'/60'/0'/0/{account_index}
```

```
Listening on localhost:8545
```

(2) 开发框架

本案例使用Truffle开发工具。Truffle是基于以太坊的智能合约开发工具，支持对合约代码的单元测试，非常适合测试驱动开发。同时内置了智能合约编译器，只要使用脚本命令就可以完成合约的编译、部署、测试等工作，大大简化了合约的开发生命周期。

(3) 以太坊接口

目前以太坊提供有JSON-RPC和web3.js两种接口。如果我们使用了Truffle框架，就默认使用了web3.js接口，因为Truffle包装了web3.js的一个JavaScript Promise框架ether-pudding，可以非常方便地使用JavaScript代码异步调用智能合约中的方法。

2. 总体架构

本案例的系统架构如图8.2所示，底层使用以太坊区块链，本地使用TestRPC来开启以太坊，通过Truffle工具，把智能合约部署在以太坊上。积分系统使用web3.js接口来调用智能合约中的方法。用户可以使用前端页面来非常方便地使用积分系统中的功能。

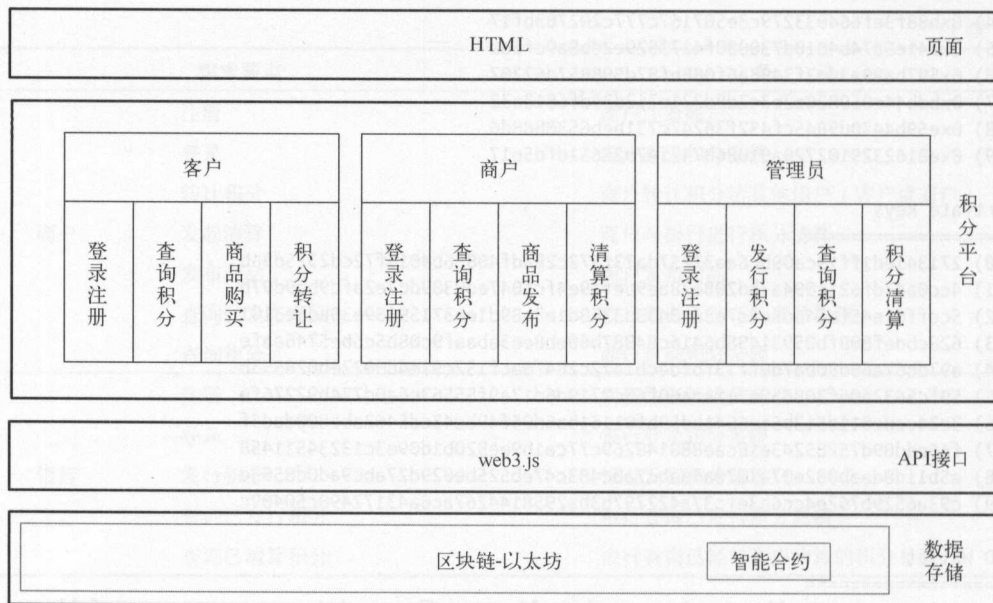


图8.2 系统架构图

8.1.4 智能合约设计

以太坊智能合约可以使用多种语言来编写，如Solidity、Serpent、LLL等，目前官方推荐使用Solidity。智能合约的设计一般有两种方案：第一种方案就是项目中的一个实体对应一个合约，这样项目中可能就会有多个合约，比如对客户实体、商户实体、银行实体分别设计3个合约，这样比较符合面向对象的思想；另一种方案是只设计一个合约，不同的对象通过结构体和映射的方式存储在一个合约中。相对来说，第二种方案较为容易理解，测试较为简单，后续的扩展维护也较为方便，因此本案例使用第二种方案。

1. 工具合约

在该案例中，因为合约会不断与前端页面交互，涉及一些数据类型转换，前端传进来的常常是string类型，而在合约中使用bytes32较多，所以要在合约中处理string和bytes32的相互转化。这里我们建立一个工具类合约，之后的工具方法可以直接加入该合约，然后让真正的主合约继承这个工具类合约即可：

```
contract Utils {
    //string类型转化为bytes32类型
    function stringToBytes32(string memory source) constant
    internal returns (bytes32 result) {
        assembly {
            result := mload(add(source, 32))
        }
    }
}
```



```

    }
    //bytes32类型转化为string类型
    function bytes32ToString(bytes32 x) constant internal returns
    (string) {
        bytes memory bytesString = new bytes(32);
        uint charCount = 0;
        for (uint j = 0; j < 32; j++) {
            byte char = byte(bytes32(uint(x) * 2 ** (8 * j)));
            if (char != 0) {
                bytesString[charCount] = char;
                charCount++;
            }
        }
        bytes memory bytesStringTrimmed = new bytes(charCount);
        for (j = 0; j < charCount; j++) {
            bytesStringTrimmed[j] = bytesString[j];
        }
        return string(bytesStringTrimmed);
    }
}

```

2. 合约状态设计

目前合约中的对象有客户、商户、管理员和商品。由于只有一个主合约，我们把管理员作为该主合约的“拥有者”，把管理员的状态作为这个合约的公共状态：

```

address owner; //合约的拥有者，银行
uint issuedScoreAmount; //银行已经发行的积分总数
uint settledScoreAmount; //银行已经清算的积分总数

```

客户、商户和商品使用struct结构体进行封装，把这些对象的属性加入结构体中。客户有账户地址、密码、积分余额、已购买的商品数组4种属性；商户有账户地址、密码、积分余额、已发布的商品数组4种属性；商品有ID、价格、所属的商户地址3种属性。

```

struct Customer {
    address customerAddr; //客户 address
    bytes32 password; //客户 密码
    uint scoreAmount; //积分余额
    bytes32[] buyGoods; //购买的商品数组
}
struct Merchant {
    address merchantAddr; //商户 address
    bytes32 password; //商户 密码
    uint scoreAmount; //积分余额
    bytes32[] sellGoods; //发布的商品数组
}
struct Good {
    bytes32 goodId; //商品ID;
    uint price; //价格;
    address belong; //商品属于哪个商户 address;
}

```

合约中应该建立一种映射，通过账户地址可以查找到客户和商户，或者通过ID找到商品。

Solidity提供了这种映射的键值对的查找方式：

```
mapping (address=>Customer) customer; //根据客户的address查找某个客户
mapping (address=>Merchant) merchant; //根据商户的address查找某个商户
mapping (bytes32=>Good) good; //根据商品ID查找该件商品
```

同时建立客户、商户和商品数组，存储所有的已注册或已添加的对象：

```
address[] customers; //已注册的客户数组
address[] merchants; //已注册的商户数组
bytes32[] goods; //已经上线的商品数组
```

3. 合约方法设计

合约的方法设计主要是针对每个功能模块中对外提供的方法进行设计，包括客户/商户注册方法、判断是否注册的方法、商户/客户登录方法、转让积分方法等。

(1) 构造方法

每个合约会有一个默认的构造函数，构造函数会在合约被初始化的时候调用。我们也可以重写构造方法，对参数做初始化操作，该案例把合约的调用者作为银行管理员的账户地址，重写构造方法如下：

```
//构造函数
function Score() {
    owner = msg.sender;
}
```

(2) 客户/商户注册

智能合约中有两种类型的方法：交易方法和constant方法。交易方法会对区块链上的状态变量进行修改，会在区块上产生一次真正的交易记录。constant方法一般用作获取变量的操作，不会对变量修改，也不会区块上产生交易记录，一般get方法都是constant方法。注册客户方法应该是一个交易方法，并使用event事件把值返回。在使用web3.js接口时，交易方法无法直接使用returns返回值，默认的回值是交易哈希，所以我们只能使用event发送事件的方式返回值。与此相反，constant方法可以使用returns直接返回数据，所以constant方法一般不写event事件。客户/商户注册实现如下：

```
//注册一个客户
event NewCustomer(address sender,bool isSuccess,string message);
function newCustomer(address _customerAddr, string _password) {
    //判断是否已经注册
    if(!isCustomerAlreadyRegister(_customerAddr)) {
        //还未注册
        customer[_customerAddr].customerAddr = _customerAddr;
        customer[_customerAddr].password = stringToBytes32(_password);
        customers.push(_customerAddr);
        NewCustomer(msg.sender, true, "注册成功");
        return;
    }else {
        NewCustomer(msg.sender, false, "该账户已经注册");
    }
}
```

```

    return;
}
}
//注册一个商户
event NewMerchant(address sender,bool isSuccess,string message);
function newMerchant(address _merchantAddr, string _password) {
    //判断是否已经注册
    if(!isMerchantAlreadyRegister(_merchantAddr)) {
        //还未注册
        merchant[_merchantAddr].merchantAddr = _merchantAddr;
        merchant[_merchantAddr].password = stringToBytes32(_password);
        merchants.push(_merchantAddr);
        NewMerchant(msg.sender, true, "注册成功");
        return;
    }else {
        NewMerchant(msg.sender, false, "该账户已经注册");
        return;
    }
}
}

```

(3) 判断客户/商户是否注册

有些方法只需要在合约内部调用，对外部接口是不可见的，可以使用`internal`关键字修饰这类方法。由于要防止客户/商户的同一账号重复注册，应该在每一次注册之前进行判断，判断是否注册的方法如下：

```

//判断一个客户是否已经注册
function isCustomerAlreadyRegister(address _customerAddr)
internal returns(bool) {
    for(uint i = 0; i < customers.length; i++) {
        if(customers[i] == _customerAddr) {
            return true;
        }
    }
    return false;
}
//判断一个商户是否已经注册
function isMerchantAlreadyRegister(address _merchantAddr)
internal returns(bool) {
    for(uint i = 0; i < merchants.length; i++) {
        if(merchants[i] == _merchantAddr) {
            return true;
        }
    }
    return false;
}
}

```

(4) 客户/商户登录

该合约案例中，使用智能合约方法获得登录对象的密码，判断是否登录成功的逻辑在JavaScript代码中进行，Solidity的方法中直接可以使用`return`返回多个值。在合约中获得登录者密码方法实现如下：

```

//查询用户密码
function getCustomerPassword(address _customerAddr)constant
returns(bool, bytes32) {
    //先判断该用户是否注册
    if(isCustomerAlreadyRegister(_customerAddr)) {
        return (true, customer[_customerAddr].password);
    }else {
        return(false, "");
    }
}
//查询商户密码
function getMerchantPassword(address _merchantAddr)constant
returns(bool, bytes32) {
    //先判断该商户是否注册
    if(isMerchantAlreadyRegister(_merchantAddr)) {
        return (true, merchant[_merchantAddr].password);
    }else {
        return(false, "");
    }
}
}

```

(5) 发行积分

在本案例中, 银行管理员可以向任何一位客户发行积分, 已发行的积分数额记录在合约的 `issuedScoreAmount` 变量中, 客户积分增长相应的数额。方法实现如下:

```

//银行发送积分给客户, 只能被银行调用, 且只能发送给客户
event SendScoreToCustomer(address sender, string message);
function sendScoreToCustomer(address _receiver, uint _amount) {
    if(isCustomerAlreadyRegister(_receiver)) {
        //已经注册
        issuedScoreAmount += _amount;
        customer[_receiver].scoreAmount += _amount;
        SendScoreToCustomer(msg.sender, "发行积分成功");
        return;
    }else {
        //还没注册
        SendScoreToCustomer(msg.sender, "该账户未注册, 发行积分失败");
        return;
    }
}
}

```

(6) 转让积分

积分可以在任意两个账户之间实现转让, 这里用同一个合约方法实现。由于需要判断调用者是客户还是商户, 参数 `_senderType=0` 表示积分发送者是客户, `_senderType=1` 表示积分发送者是商户。方法实现如下:

```

//两个账户转让积分, 任意两个账户之间都可以转让, 客户商户都调用该方法
//senderType表示调用者类型, 0表示客户, 1表示商户
event TransferScoreToAnother(address sender, string message);
function transferScoreToAnother(uint _senderType, address _sender
, address _receiver, uint _amount) {

```



```

string memory message;
if(!isCustomerAlreadyRegister(_receiver)&& !isMerchantAlreadyRegister(_receiver)) {
    //目的账户不存在
    TransferScoreToAnother(msg.sender, "目的账户不存在, 请确认后再转让!");
    return;
}
if(_senderType == 0) {
    //客户转让
    if(customer[_sender].scoreAmount >= _amount) {
        customer[_sender].scoreAmount -= _amount;
        if(isCustomerAlreadyRegister(_receiver)) {
            //目的地址是客户
            customer[_receiver].scoreAmount += _amount;
        }else {
            merchant[_receiver].scoreAmount += _amount;
        }
        TransferScoreToAnother(msg.sender, "积分转让成功!");
        return;
    }else {
        TransferScoreToAnother(msg.sender, "你的积分余额不足, 转让失败!");
        return;
    }
}
}else {
    //商户转让
    if(merchant[_sender].scoreAmount >= _amount) {
        merchant[_sender].scoreAmount -= _amount;
        if(isCustomerAlreadyRegister(_receiver)) {
            //目的地址是客户
            customer[_receiver].scoreAmount += _amount;
        }else {
            merchant[_receiver].scoreAmount += _amount;
        }
        TransferScoreToAnother(msg.sender, "积分转让成功!");
        return;
    }else {
        TransferScoreToAnother(msg.sender, "你的积分余额不足, 转让失败!");
        return;
    }
}
}
}

```

(7) 发布商品

商户可以向合约中增加一件商品, 每件商品用ID来标识, 不能重复添加相同ID。被添加的商品会使用mapping映射增加对象, 并加入商户属性的sellGoods数组中。方法实现如下:

```

//商户添加一件商品
event AddGood(address sender, bool isSuccess, string message);
function addGood(address _merchantAddr, string _goodId, uint _price) {
    bytes32 tempId = stringToBytes32(_goodId);
    //首先判断该商品ID是否已经存在
    if(!isGoodAlreadyAdd(tempId)) {
        good[tempId].goodId = tempId;
        good[tempId].price = _price;
    }
}

```

```

        good[tempId].belong = _merchantAddr;
        goods.push(tempId);
        merchant[_merchantAddr].sellGoods.push(tempId);
        AddGood(msg.sender, true, "创建商品成功");
        return;
    } else {
        AddGood(msg.sender, false, "该件商品已经添加, 请确认后操作");
        return;
    }
}

```

(8) 购买商品

客户可以输入商品ID来购买一件商品, 如果拥有的积分额度大于等于商品所需的积分, 则购买商品成功, 否则购买失败。购买成功后, 会把商品ID加入到客户的buyGoods数组中。方法实现如下:

```

//用户用积分购买一件商品
event BuyGood(address sender, bool isSuccess, string message);
function buyGood(address _customerAddr, string _goodId) {
    //首先判断输入的商品ID是否存在
    bytes32 tempId = stringToBytes32(_goodId);
    if(isGoodAlreadyAdd(tempId)) {
        //该件商品已经添加, 可以购买
        if(customer[_customerAddr].scoreAmount < good[tempId].price) {
            BuyGood(msg.sender, false, "余额不足, 购买商品失败");
            return;
        } else {
            //对这里的方法抽取
            customer[_customerAddr].scoreAmount -=
            good[tempId].price;
            merchant[good[tempId].belong].scoreAmount +=
            good[tempId].price;
            customer[_customerAddr].buyGoods.push(tempId);
            BuyGood(msg.sender, true, "购买商品成功");
            return;
        }
    } else {
        //没有这个ID的商品
        BuyGood(msg.sender, false, "输入商品ID不存在, 请确定后购买");
        return;
    }
}

```

8.1.5 系统实现

以上我们进行了总体设计和智能合约设计, 下面就进行系统实现。创建项目主要是用Truffle来进行构建, 详细实现是在8.1.4节合约方法设计之后, 使用web3.js接口实现与合约方法对接, 与设计方案中的方法接口一一对应。

1. 创建项目

本案例项目是使用Truffle框架来构建的，首先需要新建一个文件夹，然后使用终端命令行进入该文件夹，执行truffle init命令，此时就会自动创建一个基于Truffle的以太坊去中心化应用。项目创建完成后的目录结构如图8.3所示。

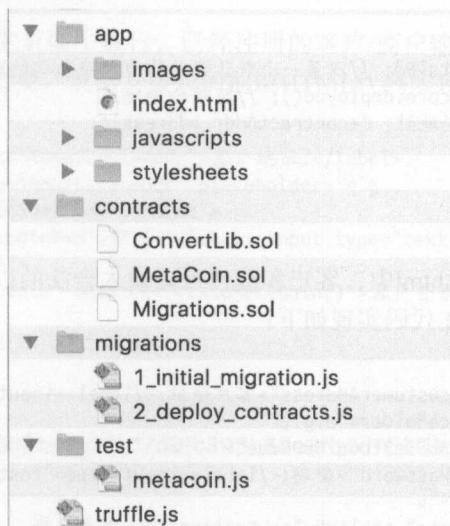


图8.3 项目创建目录结构

app文件夹中主要包含前端页面和JavaScript代码，我们实现的主要代码都会在其中的javascripts文件夹中；contracts文件夹包含了智能合约，Truffle默认会生成3个合约，我们自己实现的合约也会放入这个文件夹；migrations文件夹是关于合约部署配置的，在把合约部署到以太坊之前需要修改里面的2_deploy_contracts.js文件；test文件夹用于写智能合约的测试代码；truffle.js是整个项目的配置文件，如果需要增加JavaScript文件或者HTML页面，则需要修改truffle.js文件。

2. 详细实现

详细实现需要编写的JavaScript代码包括连接以太坊、客户/商户注册接口、客户/商户登录接口、转让积分接口等。

(1) 连接以太坊

应用需要在页面启动的时候获得部署在以太坊上的合约实例，然后才能去调用合约内的方法。由于Truffle已经默认集成了web3.js接口，所以我们可以直接使用web3.eth下的所有方法。首先在window.onload方法中获取当前以太坊上可用的账户和合约实例，在app.js中实现如下：

```

window.onload = function() {
  web3.eth.getAccounts(function(err, accs) {
    if (err !== null) {
      //如果没有开启以太坊客户端（TestRPC、geth私有链），则无法获取账号
    }
  })
}

```

```

        alert("无法连接到以太坊客户端...");
        return;
    }
    if (accs.length === 0) {
        //没有以太坊账号
        alert("获得账号为空");
        return;
    }
    accounts = accs;
    account = accounts[0]; //以第一个默认账号作为调用合约的账号
    contractAddr = Score.deployed(); //获得合约地址
    console.log("合约地址: "+contractAddr.address);
});
};

```

(2) 客户/商户注册

注册页面在主页面index.html中, 客户和商户可以输入合法的以太坊账户地址和密码来注册一个账户。注册的前端HTML代码实现如下:

```

<!-- 客户注册 -->
<br><br><br><label for="customerAddress">客户地址:</label><input type="text"
id="customerAddress" placeholder="e.g.,
0x93e66d9baea28c17d9fc393b53e3fbdd76899dae">
<br><label for="customerPassword">密码:</label><input type="text" id="customerPassword"
placeholder="e.g., *****">
<br><button id="newCustomer" onclick="newCustomer()">客户注册
</button>

```

客户注册的简单前端页面如图8.4所示。

客户地址:

e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae

密码:

e.g., *****

客户注册

图8.4 客户注册前端页面

在app.js文件中进行逻辑代码的实现如下:

```

//注册一个客户: 需要指定gas, 默认gas值会出现out of gas
function newCustomer() {
    var address=document.getElementById("customerAddress").value;
    var password = document.getElementById("customerPassword").value;
    contractAddr.newCustomer(address, password, {from: account,
        gas: 1000000}).then(function () {
        var eventNewCustomer = contractAddr.NewCustomer();
        eventNewCustomer.watch(function (error, event) {
            console.log(event.args.message);

```



```

        alert(event.args.message);
        eventNewCustomer.stopWatching(); //一定要停止监听
    });
}

```

(3) 客户/商户登录

登录操作同样在index.js主页面中进行,已经注册的客户/商户输入正确的账户密码可以成功登录,并跳转到客户页面或者商户页面。登录的前端HTML代码实现如下:

```

<!-- 客户登录 -->
<br><br><br><label for="customerLoginAddr">客户地址:</label>
<input type="text" id="customerLoginAddr" placeholder="e.g.,
0x93e66d9baea28c17d9fc393b53e3fbdd76899dae">
<br><label for="customerLoginPwd">密码:</label><input type="text" id="customerLoginPwd"
placeholder="e.g., *****">
<br><button id="customerLogin" onclick="customerLogin()">客户登录</button>

```

客户登录的简单前端页面如图8.5所示。

客户地址:

e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae

密码:

e.g., *****

客户登录

图8.5 客户登录前端页面

在app.js中进行逻辑代码的实现,根据输入的账户地址去区块链上查找指定账户的密码,并和输入密码对比是否匹配。代码实现如下:

```

//客户登录
function customerLogin() {
    var address = document.getElementById("customerLoginAddr").value;
    var password = document.getElementById("customerLoginPwd").value;
    contractAddr.getCustomerPassword(address, {from: account}).then(function(result) {
        console.log(password);
        console.log(hexCharCodeToStr(result[1]));
        if(result[0]){
            //查询密码成功
            if(password.localeCompare(hexCharCodeToStr(result[1])) === 0) {
                console.log("登录成功");
                //跳转到用户界面
                location.href="customer.html?account="+ address;
            }else {
                console.log("密码错误, 登录失败");
                alert("密码错误, 登录失败");
            }
        }
    })
}

```

```

//查询密码失败
console.log("该用户不存在, 请确定账号后再登录!");
alert("该用户不存在, 请确定账号后再登录!");
}
});
}

```

(4) 发行积分

在本案例中, 管理员可以直接使用合约调用者的地址进行登录。下面命令行界面显示的第一个地址就是调用当前合约的账户, 也是该案例默认的管理员账户。每次开启TestRPC时产生的测试账户都是不同的。可以直接使用该账号登录到管理员页面。

```

➔ ~ testrpc
Secp256k1 bindings are not compiled. Pure JS implementation will be used.
EthereumJS TestRPC v3.0.2

```

```

Available Accounts
=====
(0) 0x89af9f619b0e5394b545cecf38ced700abab10fd

```

管理员登录后会跳转到管理员页面, 需要新建一个bank.js和bank.html。之后每次新建js或html文件, 都需要按如下步骤修改truffle.js文件:

```

module.exports = {
  build: {
    "index.html": "index.html",
    "bank.html": "bank.html",
    "app.js": [
      "javascripts/app.js",
      "javascripts/bank.js"
    ],
    "app.css": [
      "stylesheets/app.css"
    ],
    "images/": "images/"
  },
  rpc: {
    host: "localhost",
    port: 8545
  }
};

```

bank.html发行积分代码实现如下:

```

<br><br><br><label for="customerAddress">客户地址:</label><input type="text"
id="customerAddress" placeholder="e.g.,
0x93e66d9baea28c17d9fc393b53e3fbdd76899dae">
<br><label for="scoreAmount">积分数量:</label><input type="text"
id="scoreAmount" placeholder="e.g., *****">
<br><button id="sendScoreToCustomer" onclick=
"sendScoreToCustomer()">发行积分</button>

```

银行发行积分的简单前端页面如图8.6所示。

客户地址:

e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae

积分数量:

e.g., *****

发行积分

图8.6 管理员发行积分前端页面

bank.js发行积分方法实现如下:

```
function sendScoreToCustomer() {
    var address=document.getElementById("customerAddress").value;
    var score = document.getElementById("scoreAmount").value;
    contractAddr.sendScoreToCustomer(address, score, {from: account});
    var eventSendScoreToCustomer = contractAddr.SendScoreToCustomer();
    eventSendScoreToCustomer.watch(function (error, event) {
        console.log(event.args.message);
        alert(event.args.message);
        eventSendScoreToCustomer.stopWatching();
    });
}
```

(5) 转让积分

登录成功后, 客户或商户可以在自己的积分额度内转让积分, 如果要转让的积分数量超过已有的额度, 则转让失败。客户-客户、客户-商户、商户-商户, 两两之间都可以进行该操作。这里以在客户中的实现为例, 新建customer.js和customer.html。

customer.html实现如下:

```
<br><br><label for="anotherAddress">转让积分地址:</label>
<input type="text" id="anotherAddress" placeholder="e.g.,
0x93e66d9baea28c17d9fc393b53e3fbdd76899dae">
<br><label for="scoreAmount">积分数量:</label><input type="text" id="scoreAmount"
placeholder="e.g., *****">
<br><button id="transferScoreToAnotherFromCustomer"
onclick="transferScoreToAnotherFromCustomer()">转让积分</button>
```

转让积分页面与银行发行积分页面基本相同。

customer.js转让积分方法实现如下:

```
//客户实现任意的积分转让
function transferScoreToAnotherFromCustomer() {
    var receivedAddr = document.getElementById("anotherAddress").value;
    var amount = parseInt(document.getElementById("scoreAmount").value);
    contractAddr.transferScoreToAnother(0, currentAccount, receivedAddr, amount, {from: account});
    var eventTransferScoreToAnother = contractAddr.TransferScoreToAnother();
```

```

eventTransferScoreToAnother.watch(function (error, event) {
    console.log(event.args.message);
    alert(event.args.message);
    eventTransferScoreToAnother.stopWatching();
});
}

```

(6) 发布商品

已登录的商户可以发布多个不同ID的商品,商品ID不能重复。商品会指定所需购买的积分额度。发布成功后,会把该件商品ID加入到商户的已购买商品数组中。在某些方法中,我们会在参数中指定gas值,其实在执行交易方法的时候,方法会自带一个默认的gas,但是如果这个方法较大,代码较多,导致默认的gas值不够,执行方法时就会发生out of gas的错误,导致交易方法失败。所以在本案例中,一些方法会显式定义gas值。以下代码分别是在merchant.html和merchant.js中的实现。

merchant.html:

```

<br><br><br><label for="goodId">商品ID:</label><input type=
"text" id="goodId"><br><label for="goodPrice">商品价格:</label>
<input type="text" id="goodPrice">
<br><button id="addGood" onclick="addGood()">添加商品</button>

```

merchant.js:

```

//商户增加一件商品
function addGood() {
    var goodId = document.getElementById("goodId").value;
    var goodPrice = parseInt(document.getElementById("goodPrice").value);
    contractAddr.addGood(currentAccount, goodId, goodPrice,
    {from: account, gas: 2000000}).then(function () {
        var eventAddGood = contractAddr.AddGood();
        eventAddGood.watch(function (error, event) {
            console.log(event.args.message);
            alert(event.args.message);
            eventAddGood.stopWatching();
        });
    });
}

```

(7) 购买商品

客户可以在自己的积分额度内通过输入商品ID购买一件商品,如果商品所需积分大于客户的积分余额,则购买失败。购买成功后,把该件商品加入到客户的已购买商品数组中。以下代码分别是在customer.html和customer.js中的实现。

Customer.html:

```

<br><br><br><label for="goodId">购买商品Id:</label><input
type="text" id="goodId">
<br><button id="buyGood" onclick="buyGood()">购买商品</button>

```


Customer.js:

```
//客户购买商品
function buyGood() {
    var goodId = document.getElementById("goodId").value;
    contractAddr.buyGood(currentAccount, goodId, {from: account,
        gas: 1000000}).then(function () {
        var eventBuyGood = contractAddr.BuyGood();
        eventBuyGood.watch(function (error, event) {
            console.log(event.args.message);
            alert(event.args.message);
            eventBuyGood.stopWatching();
        });
    });
}
```

图8.7到图8.9展示了部分功能模块实现的结果。



图8.7 客户注册成功



图8.8 银行发行积分成功



图8.9 客户转让积分成功

8.1.6 系统部署

前面我们使用truffle init命令初始化项目后,会自动生成一些不需要的示例代码,我们可以手动删除,如contracts/文件夹下面的ConvertLib.sol、MetaCoin.sol、Migrations.sol这3个文件,migrations/文件夹下的1_initial_migration.js。我们把所有的html文件直接放入app/文件夹,把所有的JavaScript文件放入app/javascripts文件夹,把写好的智能合约保存成后缀sol格式放入contracts文件夹。完成本案例后的项目目录如图8.10所示。

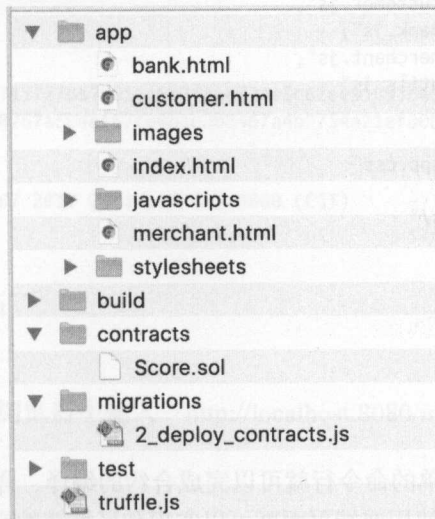


图8.10 项目完成目录结构

1. 项目配置

项目基本完成以后,需要进行简单的配置才能部署,包括合约的配置和整体配置,整体配置主要是HTML页面和JavaScript脚本的配置。

(1) 合约配置

合约需要被编译部署到以太坊客户端上,如TestRPC中,migrations/2_deploy_contracts.js是合约的部署文件,需要把我们的目标合约配置进去(已经删除默认合约的代码),2_deploy_contracts.js配置信息如下:

```
module.exports = function(deployer) {
  deployer.deploy(Score);
};
```

(2) 整体配置

由于我们创建了多个html文件和JavaScript文件,因此需要在项目根目录中的truffle.js中把这些文件注册进去。我们使用的TestRPC客户端默认使用8545端口,根据不同的需求和以太坊客户

端的配置也可以修改端口号。truffle.js实现如下：

```
module.exports = {
  build: {
    "index.html": "index.html",
    "customer.html": "customer.html",
    "bank.html": "bank.html",
    "merchant.html": "merchant.html",
    "app.js": [
      "javascripts/app.js",
      "javascripts/customer.js",
      "javascripts/bank.js",
      "javascripts/merchant.js",
      "javascripts/utils.js"
    ],
    "app.css": [
      "stylesheets/app.css"
    ],
    "images/": "images/"
  },
  rpc: {
    host: "localhost",
    port: 8545
  }
};
```

2. 项目部署

Truffle项目只需要使用简单的命令行就可以完成合约的编译，合约部署到以太坊，并把整个项目部署到本地服务器，然后使用前端页面就可以实现和区块链的交互。在部署项目之前，必须要在一个命令行终端先开启TestRPC服务，然后在另一个终端命令行进入项目根目录，分别执行以下命令。

- ❑ `truffle compile`: 编译智能合约，如果合约存在语法错误，编译将失败，并提示错误信息。如果只修改一个合约，但是项目有多个合约，想要修改后同时编译所有合约，可以执行`truffle compile --compile-all`命令。
- ❑ `truffle migrate`: 部署智能合约，把编译完成的合约部署到以太坊客户端TestRPC中。如果该命令由于异常执行失败，可以执行`truffle migrate --reset`命令。
- ❑ `truffle serve`: 开启Truffle本地服务器，默认使用8080端口，并把项目自动化部署到Truffle服务器中。

部署步骤的命令行界面实现如下：

```
➔ Score $truffle compile --compile-all
Compiling Score.sol...
Writing artifacts to ./build/contracts
➔ Score $truffle migrate --reset
Running migration: 2_deploy_contracts.js
Deploying Score...
Score: 0x0767a5b4ef67fe0b1e3b8467a4b9754a21ef80b4
```



```
Saving successful migration to network...
Saving artifacts...
→ Score $ truffle serve
Serving app on port 8080...
Rebuilding...
```

同时在另一个开启TestRPC的终端中,会打印出区块链日志,日志包括被调用的方法、交易哈希、区块号、花费的gas值等信息。TestRPC打印的命令行日志如下:

```
Listening on localhost:8545
eth_accounts
eth_sendTransaction

Transaction: 0x1fedb715727857b811a5a2f666324e5c62ee51993b46a718f1344673b3779dc1
Contract created: 0x0767a5b4ef67fe0b1e3b8467a4b9754a21ef80b4
Gas usage: 0x02e892
Block Number: 0x01
Block Time: Wed Jun 07 2017 21:13:19 GMT+0800 (CST)

eth_newBlockFilter
eth_getFilterChanges
eth_getTransactionReceipt
eth_getCode
eth_uninstallFilter
```

执行完以上步骤后,在浏览器中输入“<http://localhost:8080/index.html>”就可以访问积分系统了。

8.2 基于以太坊的电子优惠券系统案例分析

本项目将区块链技术应用到了电子优惠券系统,旨在使用区块链技术简化优惠券流通过程,消除第三方平台,实现商户真正自主地发行通用优惠券。本系统使用以太坊平台,搭建以银行和商户为节点的联盟链,通过开发的智能合约实现业务逻辑。系统切实发挥了区块链技术的P2P通信、去中介以及记录不可篡改的特性,为“优惠券”这一金融消费应用提供了一种新的解决方案。

8.2.1 项目简介

电子优惠券是目前市场上广泛应用的金融促销工具,在提升商家知名度和促进销售增长的过程中发挥了较为明显的作用,也为消费者带来了便利和优惠。但是,商家发行的电子优惠券一般都受限于有效期和发行数量,缺乏灵活的流通能力。将区块链技术应用用于通用电子优惠券,商户的自主性将会更高,不需要再依赖于中心平台,不用再支付额外的佣金或抽成;而对于用户来说,电子优惠券在手中的支配性更强,优惠券的意义也会有所提升。此外,区块链技术在领域的应用,可以将商户与用户的数据从互联网巨头的的数据垄断中解放出来,实现真正的数据自由和数据透明。

基于区块链的电子优惠券,是将区块链技术用于认定交易承诺的真实性,基于区块链技术的共识机制,电子优惠券的发行、流通和使用无需中央机构,既可以防止欺诈事件,又为商家提供了更为便利、自主、高效和低成本的服务。

本案例主要将区块链技术用作确权和记账工具,并使用智能合约记录部分系统数据,而对于系统中其他数据的管理,仍使用数据库作为存储工具。在本案例中,银行既是系统业务的参与者,也是系统的管理员,负责数据库的部署和维护,因此本系统并不是完全意义上的去中心化,但使用区块链技术将系统中的业务作为交易,在区块链上记录,同时数据库中涉及系统核心功能的数据,都可以通过区块链技术溯源认证,也切实体现了将区块链技术应用于本案例的优势。

本系统核心业务为结算券和优惠券的流通,简要流程为:银行审批商户的结算券申请,并发放相应数额的结算券;商户根据结算券余额,发行优惠券,发放给消费者;消费者使用属于自己的优惠券,在消费时向商家支付,也可以向其他消费者转赠。系统整体的流程图如图8.11所示。

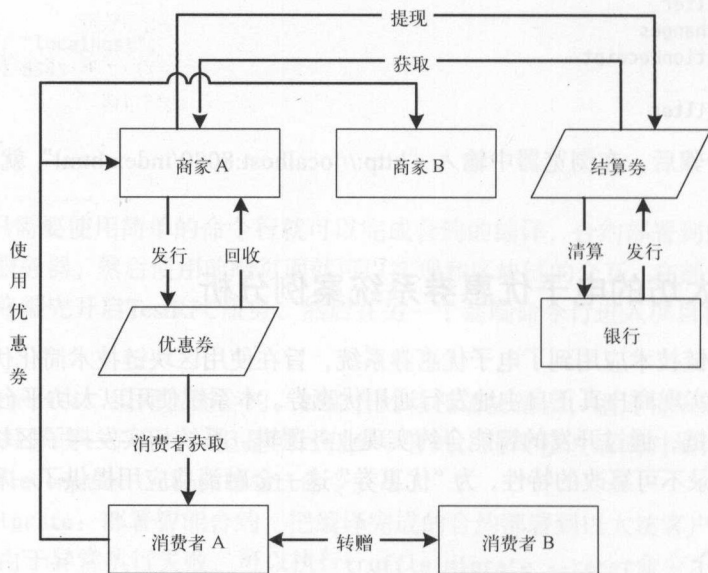


图8.11 通用优惠券系统流程图

8.2.2 系统功能分析

系统主要涉及3种用户,其中银行和商户直接交互,使用结算券作为交互媒介;商户和消费者直接交互,使用优惠券作为交互媒介。此外,银行还作为管理员对系统中的用户进行监控和管理。各个用户的具体功能如表8.2所示。

表8.2 通用电子优惠券系统功能模块表

	需求要点	备 注
消费者	注册	消费者注册账户
	登录	消费者登录系统
	申请优惠券	消费者消费后, 向商家申请优惠券
	使用优惠券支付	消费者向商家申请本次消费使用自己已有的优惠券
	查看优惠券钱包	消费者查询自己的优惠券钱包
	转赠优惠券	消费者之间互相转赠优惠券
商户	注册	商户注册账户
	登录	商户登录积分系统
	申请结算券	商户申请结算券, 作为发行优惠券的基础金额
	结算券提现	商户向银行申请将结算券提现
	查看结算券流水	商户查看结算券的申请和提现流水
	发行优惠券	商户自行制定优惠券发行规则和数量
	发放优惠券	商户同意消费者的请求优惠券的申请, 将优惠券发放给消费者
	同意消费者优惠券支付申请	商户批准消费者的优惠券使用申请
	终止发行	商户终止本次优惠券发行
	查看发行状态	商户查看当前优惠券的发行状态
银行	注册	银行注册管理员账户
	登录	银行登录管理系统
	初审、复审结算券申请	批准或拒绝商户结算券申请, 如统一申请, 则发放等额结算券
	商户注册审批	银行批准商户入链申请
	消费者管理	银行根据消费者的行为可以冻结或解冻相应的消费者账户
	优惠券查询	查询系统中所有的优惠券

8.2.3 系统总体设计

本系统的总体设计主要包括方案设计、架构设计以及底层数据存储的设计。方案设计主要明确了本系统的层次结构以及各层之间的交互设计; 架构设计部分主要分析了系统的功能架构; 此外, 由于本系统使用了数据库与区块链一同作为底层数据存储工具, 系统的底层数据存储设计部分明确了数据存储的边界, 即哪些数据使用区块链存储, 哪些数据使用数据库存储。综合上述3部分的设计分析, 一同搭建出了本系统的基本框架。

1. 方案设计

本系统业务逻辑处理部分为JavaWeb系统, 使用SSM框架整合, 为了使用区块链技术作为确权工具及实现记账功能, 本系统在教育层引入了区块链技术。系统使用geth客户端接入以太坊平台。通过工具类构建web3.js风格接口给Java后台应用层调用, 与以太坊平台进行交互, 工具类内

部实际使用Jersey框架通过JSON-RPC建立与以太坊客户端的通信。本系统在服务器上搭建以太坊私链环境进行开发测试。私链数据存储服务器上指定的文件目录下（见“系统部署”部分）。

2. 系统架构设计

本系统底层使用以太坊平台，节点通过geth客户端接入区块链网络，自行设计智能合约部署在区块链上，并在系统的应用层自己封装Web3工具类，提供web3.js风格接口（应用中可直接调用的接口），在Web3工具类中实际使用以太坊客户端的JSON-RPC接口与geth客户端交互。由于区块链在本系统中有确权作用，因此本系统将使用部署在区块链上的智能合约中存储的storage值结合数据库一同存储系统数据。

结算券与优惠券是本系统核心业务的交互媒介，二者相关的操作都作为交易，写入区块链来确权，因此本系统的智能合约应能够覆盖结算券申请、发行和提现等相关操作和优惠券发行、发放和使用等相关操作。由于区块链的不可篡改特性，系统中能反映当下状态的数据应该从区块链中读取。

系统除了核心业务数据外，还有用户的注册信息、合约对应的实体用户或者对象在区块链上的身份凭证和区块链上的合约地址，这些信息应使用数据库来存储。同时，为方便银行以管理员身份监管系统用户和运行状态，银行对于系统中的历史记录性质数据，直接从区块链上读取。而这些信息都可以通过区块链技术来溯源，因此真实性仍然可以得到保证。系统整体架构如图8.12所示。

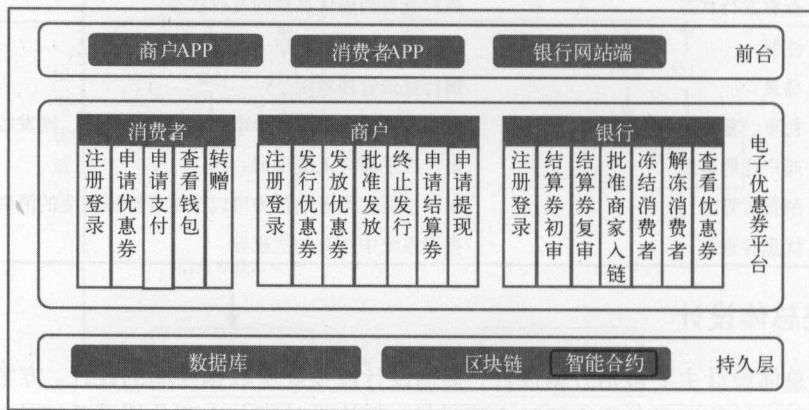


图8.12 银行通用电子优惠券系统架构图

3. 系统数据存储划分

数据读取包括从数据库和区块链两种持久化存储中读取。其中，历史流水、账单类的数据（如商户的结算券申请和提现流水、银行查询所有的系统中出现过的优惠券，等等），都采用数据库数据进行显示；对于交易相关的状态数据和系统当前的状态值类数据（如商户的当前结算券余额、当前正在发行的优惠券相关信息、消费者拥有的优惠券，等等），都从区块链上读取。本项目中两者的分类如表8.3所示。

表8.3 电子优惠券系统数据存储分类

从数据库中获取的数据	从区块链上获取的数据
商户的结算券申请与提现流水；	商户管理的当前发行中的所有优惠券信息；
银行管理系统读取的商户列表、消费者列表；	商户的结算券余额；
银行管理系统读取所有的优惠券；	消费者的钱包中的优惠券；
系统用户登录系统时的身份认证信息以及注册时的详细信息；	优惠券发行、发放以及使用过程中的优惠券详细信息；
系统中的银行、商户、消费者和优惠券的公钥	系统中当前正发行的优惠券的详细信息

8.2.4 智能合约设计

本系统的智能合约设计分为概要设计、合约状态设计以及合约方法设计3个部分。其中概要设计主要讨论了合约的种类以及每种合约的主要职责，合约状态设计主要分析了合约中应该存储哪些状态值来保证合约功能完整性和提供必要的数据库访问功能，最后合约方法设计部分分析了哪些系统实际业务应该被区块链记账以及如何将这些业务映射为合约方法。

1. 概要设计

本案例是基于区块链技术的电子优惠券系统，首先考虑系统用户。主要用户有3种：银行职员、商户和消费者。由于用户是各种业务操作的发起者，这3种用户显然应该作为合约对象在区块链中存储。

除了系统用户外，系统中存在一个重要的参与要素——优惠券，因为优惠券的发放、支付和转赠操作均需要区块链技术确权，因此本案例中将优惠券也作为合约对象，该合约主要是被动地更改状态来配合系统的业务逻辑操作。

本系统中，银行除了用户之外，还有管理员身份，因此银行合约的部署应该在系统运行前进行，银行合约地址在项目启动后应该可以直接使用。综上，本案例系统中共有4份合约，具体的设计如表8.4所示。

表8.4 通用电子优惠券系统概要设计表

合约名称	说 明	功 能
Bank	银行合约要手动部署到区块链上，与项目部署同时进行	负责审批商户的结算券相关操作
Merchant	存储商户的创建者地址，自身合约地址，维护自己的未发放、已发放、已使用的优惠券数组，并维护一个历史优惠券数组	商户可以使用合约完成发行优惠券、终止优惠券的发行；暴露接口给银行端，完成结算券申请与结算券提现功能；存储自己的结算券余额以备查询，维护优惠券数组以备溯源
Coupon	由商户创建，存储整个生命周期的所有相关状态值	一般为被动的状态改变，通过商户或消费者的操作被动更改自身状态
Consumer	存储消费者的账户地址，当前状态以及自己钱包中的优惠券	转赠优惠券到其他消费者账户

2. 合约状态设计

本系统中, 交易相关数据应从区块链上读取, 以此来发挥区块链的确权作用。在合约设计时, 应综合考虑业务逻辑与数据记录两个方面, 具体的状态值设计如下。

- **Bank合约**: Bank合约应该维护自己批准入链的所有商户, 并将银行的公钥作为本合约的“所有者”, 具体设计如下。

```
Contract Bank{
    address owner; //存储银行的公钥
    address[] merchants; //维护所有由自己审批入链的商户合约地址
}
```

- **Merchant合约**: 商户合约自身应存储自己的公钥 (owner字段) 对外提供身份认证, 以及审批自己入链的银行合约地址, 委托银行来完成结算券的相关操作的审批 (后文方法设计中会详细说明)。

商户本身作为结算券相关操作的申请者和优惠券相关操作的审批者, 应该维护这两方面的数据。首先, 从结算券的角度, 商户应该记录自己的结算券余额; 从优惠券角度, 商家要发行优惠券, 发放优惠券, 处理使用优惠券支付请求。因此, 商家应维护当前正在发行的已发行未发放、已发放未使用、已使用的3种优惠券的数组。此外, 为保证优惠券的整个生命周期可查, 还要维护一个历史优惠券数组。

```
contract Merchant{
    address owner;
    address banker;
    uint settlementBalance;
    address[] unusedCoupons; //已发放未使用的优惠券
    address[] usedCoupons; //已使用的优惠券
    address[] notGivenCoupons; //已发行未发放的优惠券
    address[] historyCoupons; //历史优惠券

    //以下两个状态值为查询数据操作设计, 后文方法设计中会详细说明
    address[] curGrant;
    mapping(bytes32 => address[]) grantPair;
}
```

- **Consumer合约**: 消费者应存储自己的公钥 (作为owner)、当前在系统中的状态 (是否被冻结) 以及自己钱包中可用的优惠券数组。此外, 由于消费者会被冻结与解冻, 因此还要存储可以更改自己状态的银行合约地址, 将冻结与解冻操作交付给银行执行。

```
contract Consumer{
    address owner;
    address banker; //可冻结、解冻消费者的银行对应的合约地址
    uint state;
    address[] coupons;
}
```

- **Coupon合约**: 优惠券合约的数据主要用于查询和被动进行状态更改, 因此, 其状态值设

计更多考虑数据的查询和整个生命周期各个阶段的状态标识,具体的字段与对应的意义如下所示。

```
contract Coupon{
    address owner; //owner存储所有人的合约地址,发行时为商户合约地址,发放后为
                //消费者的合约地址,如果发生转赠,则owner的值应该对应更改
    address granter; //发行商户的和合约地址
    uint value; //优惠券面值
    uint limit; //发行规则中的“满”字段
    bytes32 startDate; //发行规则中规定的有效起始日期
    bytes32 endDate; //发行规则中规定的有效截止日期
    uint obtainValue; //消费者获得优惠券时的消费金额
    bytes32 obtainDate; //消费者获得优惠券的时间
    bytes32 consumeDate; //消费者使用优惠券支付的日期
    uint consumeValue; //消费者使用优惠券支付时的消费金额
    uint state; //优惠券的状态 (1为已发行, 2为已发放, 3为已使用)
}
```

3. 合约方法设计

下面我们主要从构造方法、功能性方法和读取区块链存储数据3个方面来详细介绍合约方法设计。

(1) 构造方法

在系统中,构造方法负责合约的初始化和部署,4种合约对象各自的构造方法的使用如下。

- Bank合约: 在系统运行前,通过geth客户端发起交易直接部署(具体见“项目部署”部分),获得合约地址在系统中作为常项使用。
- Merchant合约: 构造方法由Bank合约调用,在系统中对应银行同意商户入链后,银行作为交易发起人调用Merchant合约的构造方法,获得Merchant的合约地址。具体代码如下。

```
Contract Bank{
    function createMerchant(address merchantAccount) OnlyOwner{
        merchants.push(new Merchant(merchantAccount)); //传入商户公钥,创建并部署商户合约
    }
    ...
    modifier OnlyOwner{
        if(msg.sender != owner){throw;} //方法修饰符,仅当方法调用者为owner时,方法才执行
    }
}
Contract Merchant{
    function Merchant(address merchantAccount) {
        owner = merchantAccount;
        banker = msg.sender;
    }
}
```

- Consumer合约: 系统中消费者不作为节点入链,只拥有公钥和合约地址,此外还需要明确可以修改其状态的银行管理员合约账号,其构造方法如下。

```

Contract Consumer{
    function Consumer(address bankAccount){
        owner = msg.sender;
        banker = bankAccount;
        state = 1; //账户未被冻结
    }
}

```

- **Coupon合约**: Coupon合约owner字段应存储系统中持有该优惠券的用户,在商户发行优惠券时,合约被创建,其构造方法如下。

```

Contract Coupon{
    function Coupon(uint _value, uint _limit, bytes32 _startDate, bytes32 _endDate){
        value = _value;
        limit = _limit;
        owner = msg.sender;
        startDate = _startDate;
        endDate = _endDate;
        consumeValue = 0; //
        state = 1; //合约创建时,优惠券一定是发行未发放状态
    }
}

```

(2) 功能性方法

本系统中的功能性方法主要有结算券相关、优惠券相关和消费者转赠3种。

A. 结算券相关

结算券相关的功能性方法主要是银行批准商户的结算券申请,在商户中定义settlementApprove()方法,并设置方法修饰符OnlyBanker,即只能由banker地址调用,在银行中定义approve()方法,调用商户的settlementApprove()方法,具体代码如下。

```

Contract Bank{
    function approve(address merchantAddress, uint amount) OnlyOwner{
        Merchant m = Merchant(merchantAddress);
        m.settlementApprove(amount);
    }
}
Contract Merchant{
    function settlementApprove(uint amount) OnlyBanker{
        settlementBalance += amount;
    }
}

```

B. 优惠券相关

- **优惠券发行**: 商家发行优惠券,即创建相应数量的优惠券合约,并将合约地址存入notGivenCoupons数组,具体代码如下。

```

Contract Merchant{
    function issueCoupon(uint value, uint limit, uint quantity, bytes32 startDate, bytes32
    endDate) OnlyOwner{

```



```

//传入参数依次为：优惠券面值、发行规则的“满”字段，有效期的开始截止日期
if(settlementBalance>=(value*quantity)){
    for(uint i=0;i<quantity;i++){
        notGivenCoupons.push(new Coupon(value, limit, startDate, endDate));
        settlementBalance -= value; //注意在发行时相应地改变商户结算券余额
    }
}
}
}

```

- 优惠券终止发行：商户可以在发行优惠券后，随时终止优惠券发行。终止发行时，需要将notGivenCoupons数组、unusedCoupons数组和usedCoupons数组中的优惠券全部移至historyCoupons数组，并将未发放的优惠券的总额加回自己的结算券余额中。如果终止发行时发现已有已发放的优惠券超出了有效期，则应将该优惠券收回。

```

Contract Merchant{
    function terminateCoupon(bytes32 curDate) OnlyOwner {

        //对于未发放的优惠券，移至historyCoupons数组并将金额加回自己的结算券余额
        for(uint k=0;k<notGivenCoupons.length;k++){
            historyCoupons.push(notGivenCoupons[k]);
            Coupon c = Coupon(notGivenCoupons[k]);
            settlementBalance += c.getValue();
        }
        delete notGivenCoupons;

        //对于已发放未使用的优惠券，如果未过期，只移至historyCoupons数组，
        //金额不加回结算券余额；如果已过期，则将金额加回结算券数组
        for(uint i=0;i<unusedCoupons.length;i++){
            Coupon c1 = Coupon(unusedCoupons[i]);
            if(c1.getEndDate()<curDate){
                settlementBalance += c1.getValue();
            }
            historyCoupons.push(unusedCoupons[i]);
        }
        delete unusedCoupons;

        //对于已使用的优惠券，直接移至historyCoupons数组即可
        for(uint j=0;j<usedCoupons.length;j++){
            historyCoupons.push(usedCoupons[j]);
        }
        delete usedCoupons;
    }
}

```

- 优惠券发放：商户按照传入的优惠券张数从notGivenCoupons数组中从后向前取相应数目的地址，将这些地址上的Coupon合约的owner设置为发放对象（消费者），将该合约地址由notGivenCoupons数组移至unusedCoupons数组，最后使用mapping数据结构，方便Java端读取区块链上的数据（在下文“读取区块链存储数据”部分有详细介绍），具体代码如下。

```

Contract Merchant{
    function grant(address _consumer, uint quantity, bytes32 date, bytes32 mark,
        uint obtainValue) OnlyOwner {
        //传入参数依次为：发放对象的合约地址、发放张数、发放日期、标识
        //和本次发放操作对应的消费金额
        if(quantity<=notGivenCoupons.length) {
            Consumer consumer = Consumer(_consumer); //获取发放对象
            for(uint i=notGivenCoupons.length-1; i>=
                notGivenCoupons.length-quantity; i--){
                Coupon couponTemp = Coupon(notGivenCoupons[i]);
                //设置被发放的优惠券的相应信息
                couponTemp.setObtainDate(date);
                couponTemp.setState(2);
                couponTemp.setObtainValue(obtainValue);
                couponTemp.setGranter(couponTemp.getOwner());
                couponTemp.setOwner(_consumer);
                consumer.addCoupon(notGivenCoupons[i]);

                //将发放出的优惠券在数组中做移动
                unusedCoupons.push(notGivenCoupons[i]);
                curGrant.push(notGivenCoupons[i]);

                //因为变量i为uint类型，当商户最后一张优惠券发放完时，通过判断i==0跳出循环
                if(i == 0){ break;}
            }

            //把本次的优惠券发放操作存入mapping由参数mark标识，方便取值
            grantPair[mark] = curGrant;
            delete curGrant;
            notGivenCoupons.length = notGivenCoupons.length - quantity;
        }
    }
}

```

- 优惠券支付：商户将指定的优惠券从unusedCoupons数组移至usedCoupons数组（如果该优惠券已被终止发行，应将该优惠券直接移至historyCoupons数组），并从Consumer合约中将该优惠券移除，同时，将该优惠券的面值加到商户的账户余额中，具体代码如下。

```

Contract Merchant{
    function confirmCouponPay(uint consumeValue, bytes32 consumeDate,
        address couponAddr, address _consumer) OnlyOwner{
        Coupon coupon = Coupon(couponAddr);
        if(consumeValue>=coupon.getLimit()){ //如果本次消费金额可使用优惠券

            //设置该优惠券的相关状态
            coupon.setConsumeValue(consumeValue);
            coupon.setConsumeDate(consumeDate);
            coupon.setState(3);

            //执行consumer的couponPay方法，将该优惠券移出消费者的优惠券数组
            Consumer consumer = Consumer(_consumer);
            consumer.couponPay(couponAddr);
        }
    }
}

```

```

//判断该优惠券是否为当前商户正在发行的优惠券，是则移至
//UsedCoupons数组，
//否则，无论是不是当前商户发行的，还是当前商户发行但已终止的，
//都直接移至historyCoupons数组
uint i = unusedCoupons.length;
for(i=0;i<unusedCoupons.length;i++){
    if(unusedCoupons[i] == couponAddr){
        break;
    }
}
if(i!=unusedCoupons.length){
    for(uint j=i;j<unusedCoupons.length-1;j++){
        unusedCoupons[j] = unusedCoupons[j+1];
    }
    unusedCoupons.length -= 1;
    usedCoupons.push(couponAddr);
    settlementBalance += coupon.getValue();
}else{
    settlementBalance += coupon.getValue();
    Merchant m = Merchant(coupon.getGranter());
    if(m.getOwner() != owner){
        m.addToUsedCoupons(couponAddr);
    }
}
}
}
}
}
}

```

C. 消费者转赠

优惠券转赠很简单，将指定优惠券从赠出方的优惠券数组移至被赠方优惠券数组，并更改优惠券的owner字段，具体代码如下。

```

Contract Consumer{
    function transfer(address newConsumer, address _coupon){
        Coupon coupon = Coupon(_coupon);
        coupon.setOwner(newConsumer);
        Consumer to = Consumer(newConsumer);
        to.addCoupon(_coupon);
        uint i = 0;
        for(;i<coupons.length;i++){
            if(coupons[i] == _coupon){
                break;
            }
        }
        for(uint j=i;j<coupons.length-1;j++){ //将该优惠券对应位置后面的数组元素，
                                                //每一项前移一位
            coupons[j] = coupons[j+1];
        }
        coupons.length -= 1;
    }
}

```

(3) 读取区块链存储数据

读取区块链存储数据主要有以下两种方法。

- 通过getter方法直接读取数据，格式如下。

```
//其中constant字段指出该方法为单纯地读取数据，不需要作为交易调用方法
function getValue() constant returns(dataType){
    return value;
}
```

- 对于系统中的特定场景，需要使用一些技巧来获取数据。在银行批准商家入链时，银行端发送交易创建商户合约，并将合约的地址存入自己维护的merchants数组，在读取本次新创建的商户合约的地址时，不能简单地读取数组最后一个字段，而是需要通过传入当前商户公钥，遍历merchants数组，找出公钥匹配的商户并返回，具体设计如下。

```
Contract Bank{
    function getCorrespondingMerchant(address merchantAccount) constant returns(address){
        uint i = merchants.length;
        for(i=merchants.length-1;i>=0;i--){
            Merchant m = Merchant(merchants[i]);
            if(merchantAccount == m.getOwner()){
                break;
            }
        }
        return merchants[i];
    }
}
```

同样，对于本次发放优惠券操作，为方便获得本次发放的所有优惠券，使用mapping数据结构，用传入的mark字段来标识本次发放的优惠券的数组（代码见前文“优惠券相关”部分）。

8.2.5 系统实现与部署

本节首先对本系统的整体部署情况进行了说明，接着给出了系统运行需要的软硬件环境，最后详细介绍了如何为本系统搭建区块链环境。

1. 系统部署图

系统部署图如图8.13所示。

本案例区块链底层基于以太坊平台，银行和商户可以作为节点接入以太坊平台，消费者通过银行节点接入以太坊。在商户用户中，总店可以建立自己的以太坊节点，分店可以通过总店接入，加盟店可以分别自行接入。如果以太坊节点使用了多台服务器，则这些服务器应处于同一网段，对外提供统一接口，确保服务器间通信无阻。另外，对于每一个商户节点而言，只能获得区块链上自己的交易信息，但所有交易信息底层都处于以太坊平台上。

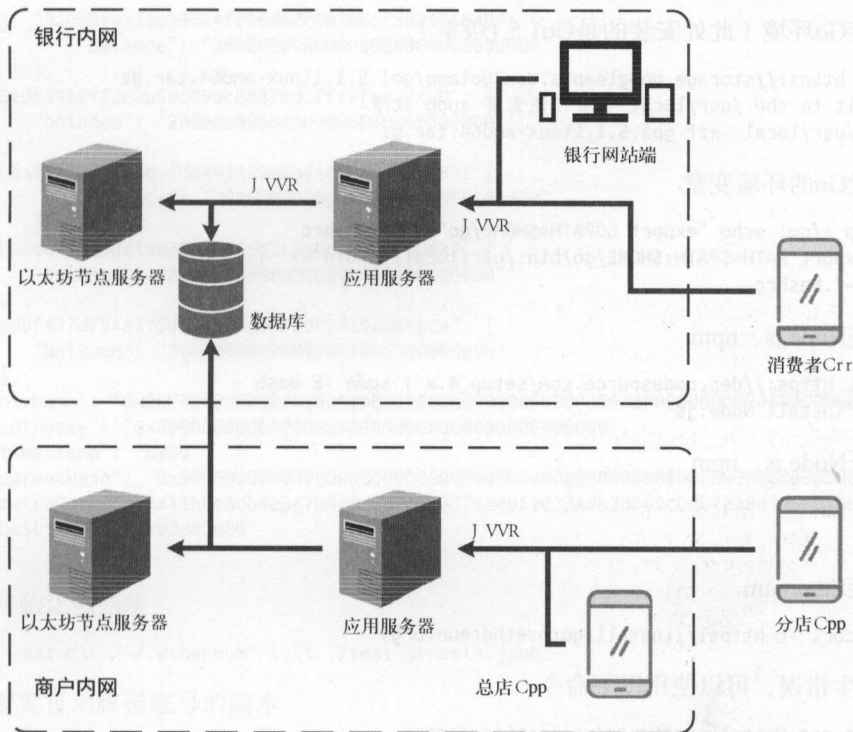


图8.13 系统部署图

2. 部署的软硬件环境

(1) 软件环境

服务端：Linux操作系统（Ubuntu 14.04）；Tomcat 8服务器软件；MySQL 5.7.6数据库管理系统以上；JDK 1.7。

客户端：安装Chrome或者Firefox浏览器。

移动端：苹果手机或iPad。

(2) 硬件环境

阿里云服务器、双核CPU、4GB内存、500GB可用存储空间。

3. 区块链环境搭建

对于一台初始化的服务器，可按照以下步骤搭建区块链环境。

(1) 安装curl命令

```
apt-get update
apt-get install git
apt-get install curl
```

(2) 安装Go环境（此处安装的是Go1.5.1版本）

```
curl -O https://storage.googleapis.com/golang/go1.5.1.linux-amd64.tar.gz
Unpack it to the /usr/local （有可能需要 sudo 权限）
tar -C /usr/local -xzf go1.5.1.linux-amd64.tar.gz
```

(3) 配置Go的环境变量

```
mkdir -p ~/go; echo "export GOPATH=$HOME/go">> ~/.bashrc
echo "export PATH=$PATH:$HOME/go/bin:/usr/local/go/bin">> ~/.bashrc
source ~/.bashrc
```

(4) 安装Node.js、npm

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -
apt-get install Node.js
```

(5) 验证Node.js、npm

```
Node.js -v
npm -v
```

(6) 安装Ethereum

```
bash <(curl -L https://install-geth.ethereum.org)
```

如果发生错误，可以使用以下命令：

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo add-apt-repository -y ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install ethereum
```

(7) 安装solc

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
which solc
```

(8) 创建账户（公钥）

在控制台输入以下命令3次，可以创建3个账号。

```
geth account new
```

(9) 编写创世块文件

在根目录（~/）下创建test-genesis.json文件。注意，可以通过设置alloc中的账号地址给你刚刚申请的公钥分配足够多的余额。

```
{
  "nonce": "0x000000000000000042",
  "difficulty": "0x1",
  "alloc": {
```

```

    "3ae88fe370c39384fc16da2c9e768cf5d2495b48": {
      "balance": "20000009800000000000000000000000"
    },
    "81063419f13cab5ac090cd8329d8fff9feed4a0": {
      "balance": "20000009800000000000000000000000"
    },
    "9da26fc2e1d6ad9fdd46138906b0104ae68a65d8": {
      "balance": "20000009800000000000000000000000"
    },
    "bd2d69e3e68e1ab3944a865b3e566ca5c48740da": {
      "balance": "20000009800000000000000000000000"
    },
    "ca9f427df31a1f5862968fad1fe98c0a9ee068c4": {
      "balance": "20000009800000000000000000000000"
    }
  },
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "timestamp": "0x00",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbbd7a38e1e50b1b82fa",
  "gasLimit": "0xb2d05e00"
}

```

(10) 初始化创始块

```
geth --datadir ~/.ethereum" init ./test-genesis.json
```

(11) 配置自动解锁账号的脚本

进入~/.ethereum目录，创建password文件，并在该文件中输入刚刚创建的每个账号对应的密码，每个密码一行，只需要输入密码即可。

(12) 编写以太坊启动脚本

创建启动脚本文件private_blockchain.sh文件，并在文件中配置如下内容：

```
geth --rpc --rpcaddr="0.0.0.0" --rpccorsdomain="*" --unlock '0,1,2' --password
~/.ethereum/password --nodiscover --maxpeers '5' --networkid '1234574' --datadir '~/.ethereum'
console
```

以后每次启动geth节点时，只需要通过以下命令即可：

```
bash private_blockchain.sh
```

(13) 部署Bank合约，并获得abi文件

手动编译合约，其中code变量为合约源码，该命令返回JSON格式的合约编译信息：

```
contracts = eth.compile.solidity(code)
```

通过以下4条取值命令分别获得4个合约对象的ABI文件：

```
bank_abi = contracts["<stdin>:Bank"]["info"]["abiDefinition"] //Bank合约的abi文件
merchant_abi = contracts["<stdin>:Merchant"]["info"]["abiDefinition"] //Bank合约的abi文件
consumer_abi = contracts["<stdin>:Consumer"]["info"]["abiDefinition"] //Bank合约的abi文件
coupon_abi = contracts["<stdin>:Coupon"]["info"]["abiDefinition"] //Bank合约的abi文件

```

创建并部署Bank合约, gas值根据实际情况设定, 该命令返回本次交易的交易哈希值:

```
txHash = eth.sendTransaction({"from": eth.accounts[0], "data": contracts[ "<stdin>:Bank"]  
["code"], "gas": "0x470000" })
```

开启矿工挖矿, 直到包含该交易的区块被接到链上:

```
miner.start()
```

交易认证后, 获取Bank合约的合约地址:

```
bankContractAddress = eth.getTransactionReceipt(txHash)
```

至此, 在运行JavaWeb项目前, 所有区块链相关的部署已全部完成。

注意, 本项目中银行、商户和消费者的以太坊账号需要通过步骤(8)所示的方法自行创建。创建好的账号目前存储在项目资源目录下的account.properties文件中, 代码中如果使用到这些账户信息, 都是从配置文件中读取的。

8.3 本章小结

本章介绍了两个基于以太坊的实际项目案例, 每个案例的介绍均包括项目简介、系统功能分析、系统总体设计、智能合约设计、系统实现和部署等, 本书提供了这些案例完整的源代码。基于前面章节所学习的以太坊基础知识和开发技术, 读者可对照本章的内容, 一步一步地动手实践, 在实战过程中更好地理解相关概念和技术, 从而为自己基于以太坊构建区块链应用项目打好基础。

企业级区块链应用实战案例详解

以太坊、Hyperledger等开源区块链平台目前还处于进一步研发和完善的过程中,在共识效率、隐私保护、大规模存储、政府监管接入等方面存在不少问题,阻碍了基于开源平台项目的应用落地和商业化进程。Hyperchain是专门针对企业级应用而设计的联盟链平台,功能完善,技术领先。目前已有多家金融机构基于Hyperchain平台开发区块链应用项目,直接对接银行系统。有部分项目已完成落地并进行商业推广。本书前面的章节已经对Hyperchain的核心原理和开发实践进行了详细梳理,本章将更加贴近实战,并介绍两个基于Hyperchain的企业级区块链应用项目案例:数字票据系统和出行打车平台。

9.1 基于 Hyperchain 的数字票据系统案例分析

本节基于某银行的业务需求设计了一个基于Hyperchain的数字票据系统。该系统主要提供账户管理、票据操作、票据查询等功能,通过调用部署在Hyperchain区块链平台的智能合约,为银行的客户端系统提供RESTful服务接口。本节设计的票据系统可以成为区块链在金融领域的范例,为其他基于区块链的应用提供一定的参考和借鉴。

9.1.1 项目简介

票据业务是银行等金融机构一项重要的资产业务,具有多方参与、流通转让性、严格标准化等特性,与联盟区块链技术的授权准入、可追溯历史以及智能合约的强控制性非常契合。基于区块链的数字票据系统,不仅可以加速票据市场统一化、数字化,而且可以保证票据业务的安全性,有效防范票据市场风险,同时可以减少清算成本,具有深远的影响和意义。

本节的票据系统针对的用户群有两大类:银行和业务客户。银行通过联盟许可便可接入区块链,因此本系统主要针对业务客户的需求进行设计。用户间的票据流转操作及生命周期如图9.1所示。

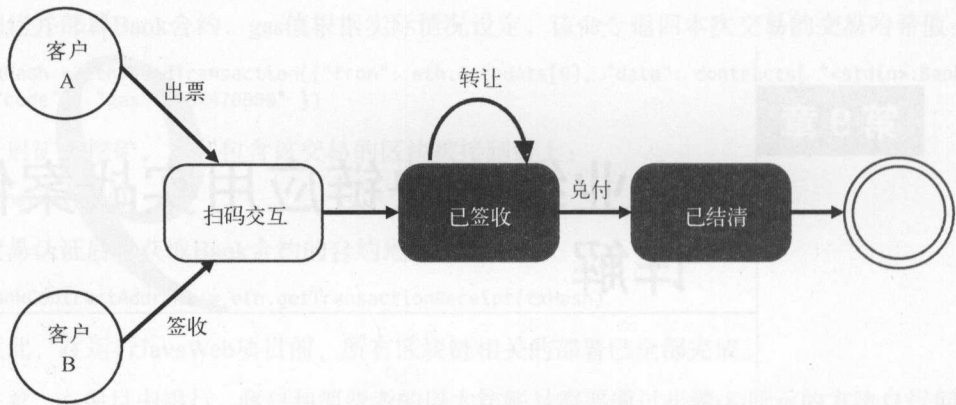


图9.1 票据生命周期

用户间的出票和签收通过扫码交付完成，一方出票，一方进行签收，交互完成后，票据状态为已签收。已签收的票据可以进行转让或兑付操作。票据兑付后状态为已结清，票据生命周期至此结束。

9.1.2 系统功能分析

某银行计划依托区块链技术，对传统柜面票据业务进行改造，研发一种创新的银行票据系列产品。本节基于该银行的业务需求，设计并实现了基于区块链的数字票据系统。

本节实现的数字票据系统基于Hyperchain联盟区块链技术开发。现假设其中一家加入联盟的银行为A银行。数字票据是一种创新型的票据，支持通过移动互联网设备完成全流程业务办理。相关业务术语和解释如表9.1所示。

表9.1 数字票据业务术语解释

术 语	解 释
数字票据	一种创新型的小金额票据，支持通过移动互联网设备完成全流程业务办理
出票	付款人签出数字票据并交付给收款人的行为
兑付	将数字票据对应的资金从银行专用内部账户兑付至持票人的银行账户
转让	未兑付的数字票据通过背书转移给其他用户的行为

本系统所涉及的票据操作具体如下。

- ❑ 出票与签收。付款人和收款人通过扫描二维码完成票据出票（签收）操作。付款人在出票过程中设定票据金额、类型、有效期等信息，并生成相应的二维码，收款人扫码后确认票据信息，并进行签收。
- ❑ 转让。转让与出票类似，也是通过扫码完成转让操作，并支持部分转让。

- 兑付。用户可对已签收的票据进行线上兑付，同一张票据支持不限次数部分兑付。若某张票据进行了拆分兑付操作，并处于中间状态（如兑付中），则拆分后剩下的票据依然支持转让、兑付等操作。

9.1.3 系统总体设计

区块链应用与传统应用最主要的区别是去中心化。基于区块链的应用本质上就是使用区块链和智能合约实现了传统应用的数据库和应用服务后台程序中的核心逻辑，降低上层业务系统的复杂度。在系统设计上，应用底层使用区块链保证共识的达成和交易的进行，通过编写智能合约实现业务逻辑，并将其部署在区块链平台。理论上，区块链平台可以直接替换传统应用的应用服务器，为客户提供服务。但是，由于智能合约对网络等操作进行了限制，如果应用复杂，涉及外部交互，则需要设计一个中间层用于对接客户端和区块链平台。传统应用的主要业务逻辑是通过应用服务后台程序实现，而本节票据系统的主要业务逻辑是在智能合约实现。该票据系统在架构上总体分为4层：应用层是系统客户端；中间层整合区块链平台接口和外部系统接口为应用层提供服务；通用接口层是提供应用支撑的外部系统接口封装；底层为区块链平台，主要业务逻辑通过智能合约实现。

1. 总体思路

本章实现的数字票据系统，通过Hyperchain区块链平台的信任机制提升了票据的信用度，同时通过移动终端App将客户从传统线下渠道引导至线上移动互联网渠道，提升了业务的便捷性。

系统的总体思路如下。

- (1) 用户通过移动App实现数字票据的相关操作，并支持账户管理、查询等功能。
- (2) 票据的流转通过智能合约实现，保证了票据交易的安全性、有效性和不可抵赖性，同时通过区块链天然清算的特性完成资金的清算。
- (3) 系统为App提供接口支持，同时对接银行核心系统、短信服务等接口完成资金流转和安全验证。

2. 应用架构

图9.2所示为系统模块图，本系统涉及数据存储模块、基础支撑模块、数字票据系统、手机App、银行核心系统、短信平台等6大模块。

(1) 数据存储模块

底层基于Hyperchain平台，利用区块链上存储的数据具有不可逆和不可修改的特性，来保证票据交易的安全性和可靠性。其中，使用可编程的智能合约来控制、约束票据的流转，并对票据进行确权。

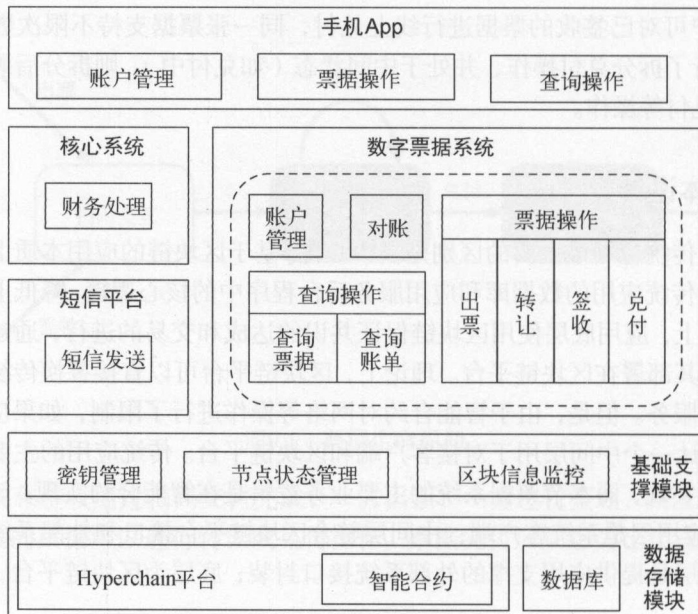


图9.2 系统模块图

(2) 基础支撑模块

基础支撑模块包括用户密钥的生成，以及对联盟链节点状态和节点中区块信息进行监控。

(3) 数字票据系统

基于区块链底层实现的数字票据系统，为企业网银和App提供账户管理和票据操作（包括出票、转让、签收、兑付）、查询操作（包括查看票据、查看账单）等接口，同时调用核心系统账务服务用于资金的流转以确认交易，调用短信平台发送短信进行身份校验。

(4) 手机App端

用户可通过手机App完成票据操作，通过调用系统接口实现账户管理、出票、转让、签收、兑付票据操作功能，以及查看票据池、账单等查询功能。

(5) 外部系统

银行核心系统和短信平台为票据系统的外部服务接口。核心系统提供银行核心转账接口。系统通过调用短信服务接口发送验证码进行身份验证。

3. 主要功能设计

本节实现的系统中，功能模块分为账户管理、票据操作、查询操作等模块，其中查询就是调用智能合约方法查询票据和账单信息，下面主要对票据操作功能模块进行操作流程和系统交互说明。

整体来讲，票据的出票、转让、兑付操作，业务逻辑大同小异，主要是针对不同用户，业务

逻辑会有所不同。下面就以最复杂的经办人出票为例，介绍其流程和系统交互关系。

如图9.3所示，用户在App端通过扫码交互，确认交易之后，由付款方点击出票，发送出票方和签收方信息到票据系统。系统校验用户合法之后，调用智能合约出票方法，完成出票。智能合约方法调用成功之后，会调用银行核心系统转账接口进行扣款操作。扣款成功之后，调用智能合约出票成功方法；如果调用失败，返回出票失败；如果调用成功，返回出票成功。若扣款失败，调用合约出票失败方法，返回出票失败，银行扣款失败。

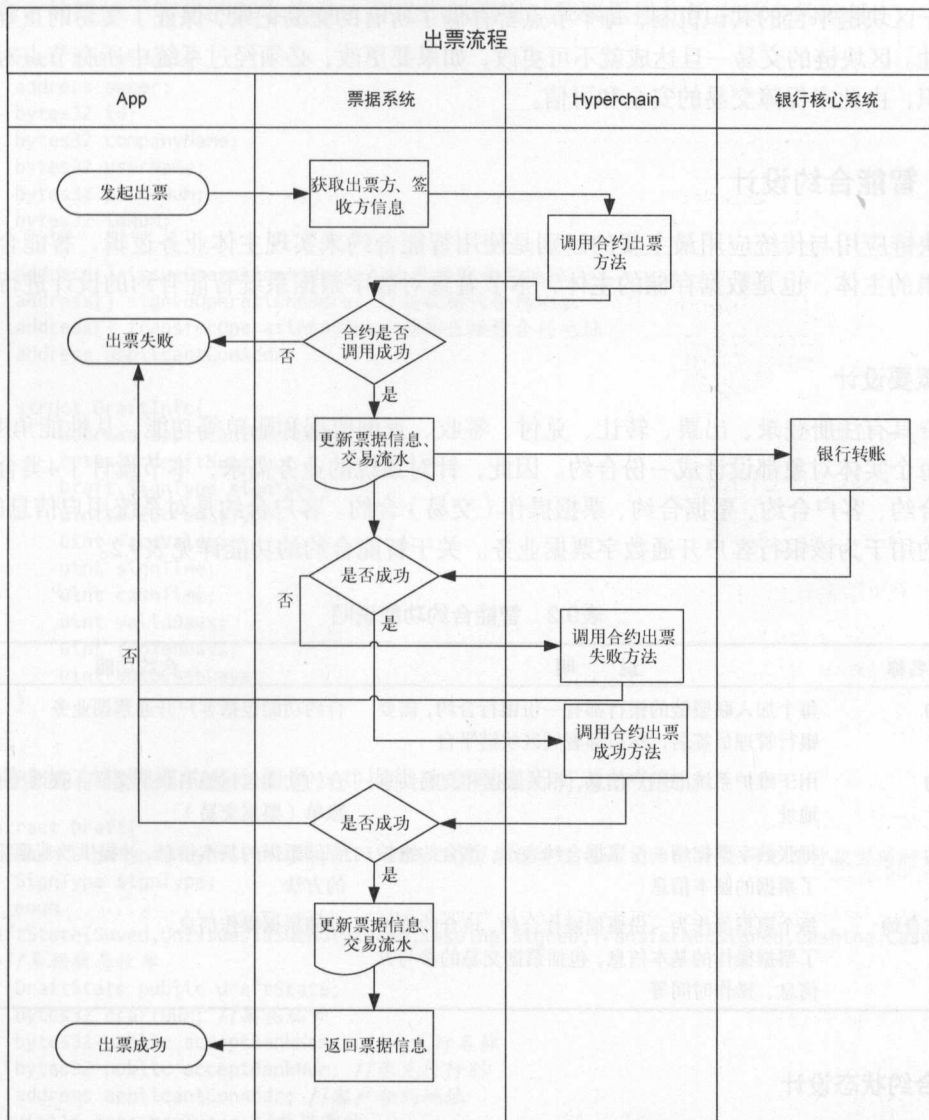


图9.3 出票流程

4. 安全性设计

本节实现的票据系统中,当用户发起一笔票据交易后,服务器将发送一份包含了交易序号的交易报文给银行内部服务器,用于处理实际的资金交易。交易序号是一笔交易的唯一标识。当接收方接收了交易报文后,需要判断报文中包含的交易序号是否已经存在于本地记录中,如果已经存在,则说明这是一份重复发送的报文,将不被系统正常处理。这个机制保障了系统不会重复处理具有相同交易序号的交易。

基于区块链平台的共识机制,每个节点都存储了所有的交易记录,保证了交易的正确性和不可篡改性。区块链的交易一旦达成就不可更改,如果要更改,必须经过系统中所有节点对其修改达成共识,由此来保障交易的安全和可信。

9.1.4 智能合约设计

区块链应用与传统应用最主要的区别是使用智能合约来实现主体业务逻辑,智能合约既是程序逻辑的主体,也是数据存储的主体,本节着重对数字票据系统智能合约的设计进行了阐述和说明。

1. 概要设计

客户具有注册登录、出票、转让、兑付、签收、查询票据和账单等功能,从性能角度考虑,本节将每个实体对象都设计成一份合约。因此,针对系统的业务需求,本节设计了4类合约,包括银行合约、客户合约、票据合约、票据操作(交易)合约。客户合约是对系统用户信息的抽象,银行合约用于为该银行客户开通数字票据业务。关于智能合约的功能详见表9.2。

表9.2 智能合约功能说明

合约名称	说 明	合约功能
银行合约	每个加入联盟链的银行都有一份银行合约,需要银行管理员签名,然后部署到区块链平台	合约功能包括客户开通票据业务
客户合约	用于维护系统的用户信息,相关票据和交易信息地址	合约功能包括签名票据交易、查询所有票据、账单(票据交易)
票据合约	每张数字票据用一份票据合约表示,该合约维护了票据的基本信息	存储票据的基本信息,并提供改变票据所有者的方法
票据操作合约	每个票据操作作为一份票据操作合约,该合约维护了票据操作的基本信息,包括票据交易的参与方信息、操作时间等	存储票据操作信息

2. 合约状态设计

银行合约维护了银行合约部署者的公钥,以及客户、票据、票据操作等相关合约地址。

```
contract Bank{
    address owner; //存储银行公钥
    bytes32 public bankName; //银行名称
    bytes32 public bankID; //银行行别

    address[] individualConAdrs; //维护所有由自己审批的客户合约地址
    address[] cashedDraftAdrs; //维护所有已兑付票据合约地址
    address[] draftOperations; //维护票据操作合约地址
}
```

客户合约维护了客户的基本信息、可操作的票据以及票据操作历史的合约地址。

```
contract Individual{
    address owner;
    bytes32 id;
    bytes32 companyName;
    bytes32 userName;
    bytes32 phoneNum;
    bytes32 idNum;

    address[] issueOperationAdrs; //出票操作合约地址
    address[] signedOperationAdrs; //签收操作合约地址
    address[] transferOperationAdrs; //转让操作合约地址
    address applicantConAddr;

    struct DraftInfo{
        address applicantConAddr;
        bytes32 draftNum;
        Draft.SignType signType;
        uint16 currencyType;
        uint faceValue;
        uint signTime;
        uint cashTime;
        uint validDays;
        uint frozenDays;
        uint autoCashDays;
    }
}
```

票据合约存储票据的基本信息，并提供改变票据所有者的方法。

```
contract Draft{
    enum SignType{PayAtSight,PayAtFixedDate} //票据类型包括见票即付型和定日付款型两种
    SignType signType;
    enum
    DraftState{Saved,UnIssue,IssueNotSigned,Issuing,Signed,TransferNotSigned,Cashing,Cashed,Invalid} //票据状态枚举
    DraftState public draftState;
    bytes32 draftNum; //票据编号
    bytes32 public acceptBankName; //承兑行名称
    bytes32 public acceptBankNum; //承兑行行别
    address applicantConAddr; //客户合约地址
    uint16 currencyType; //票据币种
    uint faceValue; //票据面额
```

```

uint signTime; //出票时间
uint signedTime; //签收时间
uint cashTime; //兑付时间
uint validDays; //有效期
uint frozenDays; //冻结期
address[] draftOperationAdrrs; //票据操作合约地址数组
}

```

票据操作合约存储票据操作信息。

```

contract DraftOperation{
    bytes32 public sequenceNum; //操作流水号
    address public fromConAddr; //交易上家合约地址
    bytes32 public fromId; //交易上家
    address public toConAddr; //交易下家合约地址
    bytes32 public toId; //交易下家
    address public draftAddr; //原票据地址
    address public newDraftAddr; //新票据地址
    uint public value; //金额
    uint public operationTime; //操作时间
    enum OperationType{Issue,Transfer,Cash} //交易类型
    OperationType public operationType;
    enum TxState{Success,ToSignature,ToCharge,SignatureFail,ChargeFail} //交易结果
    TxState public txState = TxState.ToSignature;
}

```

3. 合约方法设计

(1) 构造方法

在系统中，构造方法负责合约的初始化和部署，4种合约对象各自构造方法的使用如下所示。

```

contract Bank{
    function Bank(bytes32 _bankName,bytes32 _bankID){
        owner=msg.sender;
        bankName = _bankName;
        bankID = _bankID;
    }
}

contract Individual{
    function Individual(address _individualAddr,bytes32 _id,
bytes32 _companyName, bytes32 _phoneNum,bytes32
_userName,bytes32 _idNum){
        owner = _individualAddr;
        id=_id;
        companyName=_companyName;
        userName=_userName;
        phoneNum=_phoneNum;
        idNum=_idNum;
    }
}

contract Draft{
    function Draft(bytes32 _draftNum,SignType _signType,DraftState

```



```

_draftState,uint _validDays,uint _frozenDays, uint
_signTime,uint _faceValue,address _applicantConAddr,uint16 _currencyType){
    draftNum=_draftNum;
    signType=_signType;
    draftState=_draftState;
    validDays=_validDays;
    frozenDays=_frozenDays;
    signTime=_signTime;
    faceValue=_faceValue;
    applicantConAddr=_applicantConAddr;
    currencyType=_currencyType;
}

contract DraftOperation{
    function DraftOperation(bytes32 _sequenceNum,address
    _fromConAddr,bytes32 _fromId,address _toConAddr,bytes32
    _toId,address _draftAddr,address _newDraftAddr,uint
    _value,uint _operationTime,OperationType _operationType){
        sequenceNum=_sequenceNum;
        fromConAddr=_fromConAddr;
        fromId=_fromId;
        toConAddr=_toConAddr;
        toId=_toId;
        draftAddr=_draftAddr;
        newDraftAddr=_newDraftAddr;
        value=_value;
        operationTime=_operationTime;
        operationType=_operationType;
    }
}

```

(2) 功能性方法

基本操作：将常用功能抽象成方法，方便合约内部调用。

```

//获得操作地址
function getOperationAddr(uint i) constant returns(address){
    return draftOperationAddrs[i];
}
//获得操作地址长度
function getOperationsLen() constant returns(uint){
    return draftOperationAddrs.length;
}
//获得数字票据面额
function getFaceValue() constant returns(uint){
    return faceValue;
}
//设置数字票据状态
function setState(DraftState _state){
    draftState=_state;
}

// 改变合约所有者
function changeOwner(address _applicantConAddr,address

```

```

        _operatorConAddr){
    applicantConAddr=_applicantConAddr;
    operatorConAddr=_operatorConAddr;
}

```

出票：出票的方法由经办人或个人用户调用，在发起出票申请后，生成一张新的票据，然后调用银行核心系统进行动账操作，若动账成功则调用出票方法。

```

// 出票成功
function issueSuccess(address draftAddr,address toAppConAddr,bytes32
sequenceNum,address fromConAddr,address toConAddr,uint
operationTime) returns(address){
    Draft draft=Draft(draftAddr);
    uint value=draft.getFaceValue();

    DraftOperation draftOperation=new DraftOperation(sequenceNum,
        fromConAddr,0x0,toConAddr,0x0,draftAddr,

    draftAddr,value,operationTime,
    DraftOperation.OperationType.Issue);

    draft.setState(Draft.DraftState.Signed);
    draft.changeOwner(toAppConAddr,toConAddr);

    issueOperationAdrs.push(draftOperation);
    return draftOperation;
}

```

兑付票据：兑付申请由客户发起，在银行核心系统进行动账操作，动账成功后，调用兑付成功方法。

```

// 兑付数字票据成功
function cashSuccess(address draftAddr,address
newDraftAddr,address _bankConAddr,address
draftOperationAddr){
    Draft newDraft=Draft(newDraftAddr);
    newDraft.changeOwner(_bankConAddr,_bankConAddr);
    newDraft.setState(Draft.DraftState.Cashed);

    DraftOperation draftOperation =
    DraftOperation(draftOperationAddr);
    copyOperationAddr(draftAddr,newDraftAddr,draftOperation);
}

// 产生新数字票据时将父数字票据的所有历史记录复制进新数字票据
function copyOperationAddr(address draftAddr,address
newDraftAddr,address operation){
    Draft oldDraft=Draft(draftAddr);
    Draft newDraft=Draft(newDraftAddr);
    uint length=oldDraft.getOperationsLen();
    for(uint i=0;i<length;i++){
        address temp=oldDraft.getOperationAddr(i);
    }
}

```

9.1.5 系统实现与部署

本系统为银行提供服务，整个系统网络分为两层：内网和外网。系统网络拓扑结构如图9.4所示。App和企业网银客户端属于外网，系统应用服务器和Hyperchain平台部署在银行内网中，并通过银行数据中心前置机作为外网接入节点。

- 数字票据系统应用服务器部署在银行内网，通过F5负载均衡实现高可用性；
- Hyperchain平台部署在银行内网，通过多节点部署实现高可用；
- 使用HTTPS协议，确保网络传输安全。

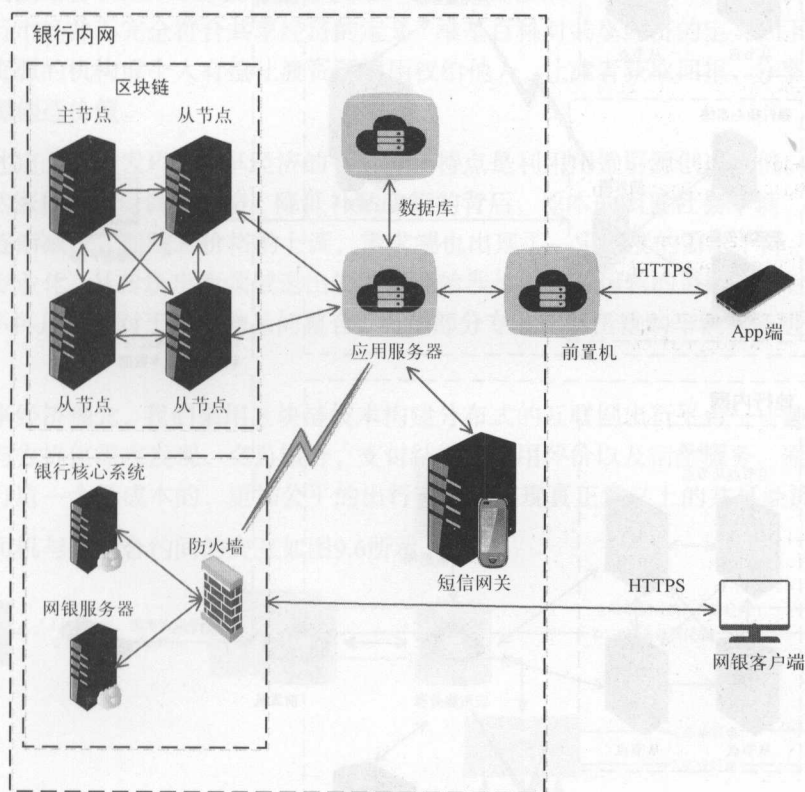


图9.4 系统网络拓扑图

当其他银行加入联盟时，对应地也需要部署Hyperchain区块链平台和本节实现的票据系统应用服务，如图9.5所示。其中，Hyperchain服务器必须能够相互通信，以满足行内区块链节点的相互通信和数据同步。不同的银行只能访问自己的用户信息和交易数据，但是账目信息都存在底层Hyperchain区块链平台上，以减少结算成本。

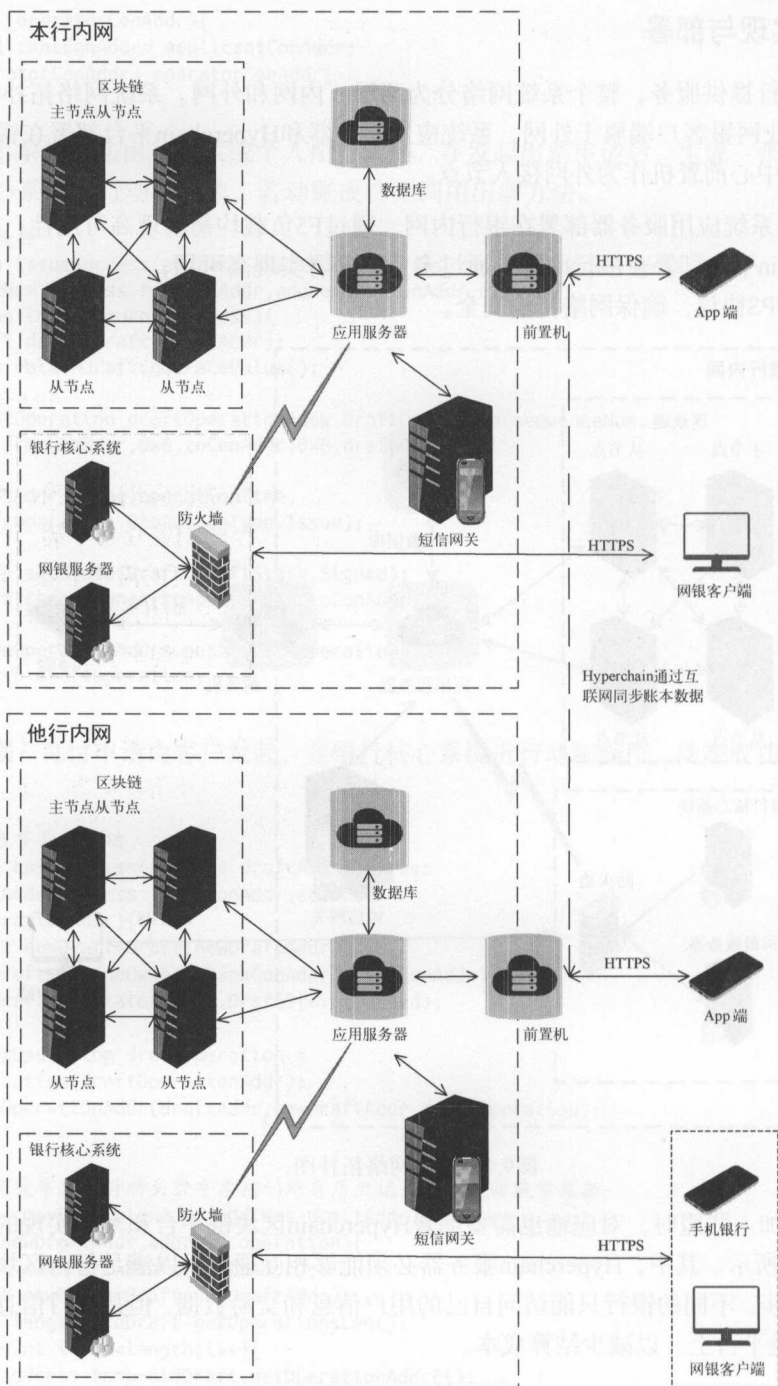


图9.5 其他银行加入联盟网络拓扑图

9.2 基于 Hyperchain 的出行打车平台案例分析

本节设计实现了一个基于 Hyperchain 的网约车平台。该平台主要提供身份管理、订单撮合、支付等功能，通过调用部署在 Hyperchain 区块链平台的智能合约，为乘客和司机提供一个去中心化的网约平台。

9.2.1 项目简介

近几年滴滴出行、Airbnb 等公司非常火爆，其背后都涉及一个叫作“共享经济”的概念。但是现在的出行市场并不完全符合共享经济的定义。维基百科对共享经济的定义如下：共享经济是指拥有闲置资源的机构或个人有偿让渡资源使用权给他人，让渡者获取回报，分享者利用分享他人的闲置资源创造价值。

通过上述定义不难发现，共享经济的一个重要特点是利用闲置资源创造价值。而如今的出行市场，在悄然涨价、自由调控价格、降低补贴政策的背后，原本的闲置社会车辆、司机在获益有限的状况下逐渐减少，而随着价格的上调，需求端也出现了一定程度的需求下降。随之而来的，是一个高度专业化、从传统出租领域退出的群体开始涉足网约车运营的道路。同时，现如今的很多车辆，也不再是单纯对于闲散物品的整合，而是部分专业化队伍重购车辆设备进行的一次行业新转型。

基于共享经济理念，我们采用区块链技术构建分布式的互联网出行平台——趣快出行。该平台为多个参与方提供需求发现、交易撮合、支付结算、信用评价以及衍生服务。通过区块链，我们将有能力打造一个低成本的、更加公平的出行市场，实现真正意义上的共享经济。

乘客、司机与智能合约间的交互如图9.6所示。

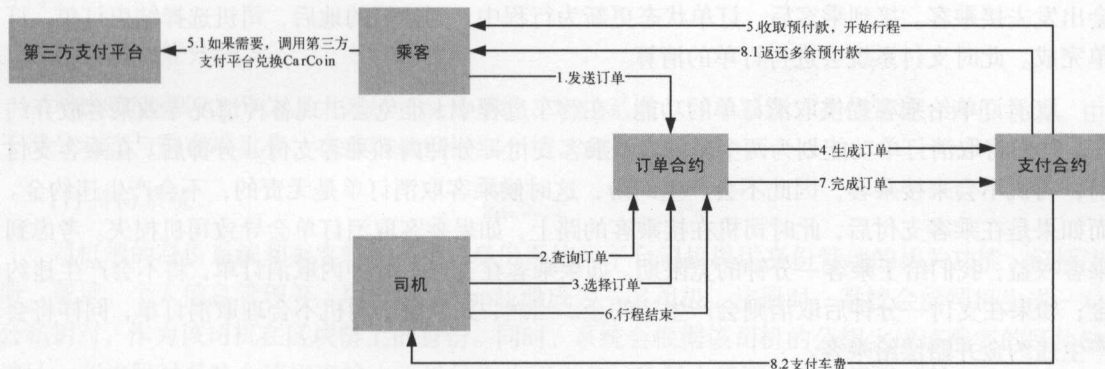


图9.6 乘客、司机与智能合约间的交互

9.2.2 系统功能分析

系统给乘客与司机提供一个P2P网约车平台，本节将分别对乘客和司机两端进行功能分析。

1. 乘客功能分析

乘客功能整体上分为五大系统：身份系统、定位系统、订单系统、支付系统和评价系统。

(1) 身份系统

身份系统为平台乘客提供身份管理的相关功能，包括注册、登录、个人信息管理等。用户通过App注册成为一名乘客。注册时，系统会给乘客生成一对公私钥对，作为该乘客在区块链上的身份。同时，系统会根据该乘客的公钥生成该乘客的区块链地址。在注册时系统会请用户输入手机号作为用户名，注册之后用户即可登录系统。

个人信息部分包括乘客姓名、手机号、头像、常用地址等基本信息。乘客登录后可以进行个人信息的完善及修改。

(2) 定位系统

趣快出行通过手机上的GPS系统给乘客提供定位服务。定位系统给订单系统提供了强有力的基础数据支持。

(3) 订单系统

订单系统是趣快出行中最重要、最关键的功能系统。该系统负责所有与订单有关的功能。整体上订单系统分为约车服务、取消订单、查询订单。

约车服务为乘客提供约车功能。乘客设置好出发地及目的地后，即可约车。当有司机接单后，用户会收到预付款请求。乘客支付完预付款后，司机会收到系统乘客已支付的通知。此时司机就会出发去接乘客。接到乘客后，订单状态更新为行程中。到达目的地后，司机选择结束订单，订单完成。此时支付系统会进行订单的清算。

取消订单给乘客提供取消订单的功能。在约车过程中，难免会出现各种情况导致乘客放弃约车。我们将取消订单功能划为两个时间段：乘客支付一分钟内和乘客支付一分钟后。在乘客支付前，司机不会来接乘客，因此不会产生纠纷，这时候乘客取消订单是无责的，不会产生违约金。而如果是在乘客支付后，此时司机在接乘客的路上，如果乘客取消订单会导致司机损失。考虑到乘客权益，我们给予乘客一分钟的宽限期。如果乘客在支付一分钟内取消订单，将不会产生违约金；如果在支付一分钟后取消则会产生违约金。当然，如果由于司机不合理取消订单，同样将会产生违约金并赔偿给乘客。

查询订单为乘客提供查询历史订单的功能。用户可以查询自己所有的历史订单，包括已完成和未完成的订单。每条订单会列出详细信息，包括出发地、目的地、时间、价格、司机、账单等。

(4) 支付系统

支付系统是趣快出行中第二关键的功能系统。该系统负责所有与支付有关的功能。整体上支付系统可以分为兑换、提现、支付预付款、订单结算几个模块。

趣快出行平台使用CarCoin进行支付相关的操作。CarCoin是为趣快出行平台发行的一种“数字现金”。

兑换模块提供兑换CarCoin的功能。用户可以通过第三方支付平台进行人民币与CarCoin的兑换。

提现模块提供CarCoin提现的功能。用户可以通过第三方支付平台将CarCoin提现成人民币。

支付预付款模块提供支付功能。趣快出行设计初期考虑司机的权益,采用预付款方式,即在订单开始前乘客即要支付订单费用。当乘客发起约车请求,并有司机接单后,乘客就会收到支付预付款请求。系统会根据订单信息计算出合理的预付款。

订单结算模块提供订单费用结算功能。通常乘客支付的预付款会多于实际产生的车费。因此,在订单结束后,需要进行结算。具体来说,就是根据实际行程计算出实际车费,支付给司机,然后将多余的车费退还给乘客。当然,如果产生预付款少于实际车费的情况,结算功能也要能够从乘客处收取不足的车费,并支付给司机。

(5) 评价系统

由于趣快出行去中心化、无人工干预的特点,评价系统显得尤为重要。趣快出行将根据乘客和司机的评分动态调整订单价格、订单分配等。

评价系统给乘客提供评价司机的功能。行程结束后,乘客根据行程状况给司机进行合理的评分。

2. 司机功能分析

司机功能整体上分为五大系统:身份系统、定位系统、订单系统、支付系统、评价系统。由于部分功能与乘客端重叠,在乘客端已详细介绍,此处不再赘述。

(1) 身份系统

司机端的身份系统和乘客端的类似。身份系统为平台司机提供身份管理的相关功能,包括注册、登录、个人信息管理等。用户通过App注册成为一名司机。注册时,系统会给司机生成一对公私钥对,作为该司机在区块链上的身份。同时,系统会根据该司机的公钥生成该乘客的区块链地址。在注册时系统会请用户输入手机号作为用户名,注册之后用户即可登录系统。

个人信息部分包括司机姓名、手机号、头像、车型、车牌号等基本信息。司机登录后可以进行个人信息的完善及修改。

(2) 定位系统

趣快出行通过手机上的GPS系统给用户提供定位服务,给订单系统提供了强有力的基础数据支持。

(3) 订单系统

司机端的订单系统分为开始接单、停止接单、开始行程、结束行程。

开始接单给司机提供“上班”的功能。司机选择开始接单后,就会进入接单状态。这时订单系统会给司机推送合适的订单供司机选择。司机选择一个订单后,系统便会通知乘客支付预付款。

停止接单给司机提供“下班”的功能。司机选择停止接单后,就会进入离线状态。订单系统不会再给司机推送订单。

开始行程给司机提供“开始工作”的功能。当司机接到乘客后,选择开始行程进入行程中。此时订单系统会记录下司机位置、行程轨迹、行程时间等行程信息。

结束行程给司机提供“结束工作”的功能。当结束行程后,司机选择结束行程。订单系统此时会进行订单的结算工作。

(4) 支付系统

司机端的支付系统分为费用结算、提现功能。

订单结算给司机提供订单费用结算功能。系统将从订单系统获取订单实际车费支付给司机。

提现给司机提供提现功能。司机可以将自己收到的CarCoin通过第三方支付平台提现成人民币。

(5) 评价系统

司机端的评价系统为司机提供给乘客评分的功能。在趣快出行平台中,给乘客的评分同样重要。司机根据行程中乘客的表现给乘客合理评分。

3. 第三方支付平台功能分析

在趣快出行平台中流通的“货币”是CarCoin,但是现在没有人民币与CarCoin的交易所。为了打通CarCoin与现实世界的联系,趣快出行需要引入第三方支付平台合作方。

第三方支付平台在给趣快出行与用户提供一个交易的平台。用户与趣快出行官方账号在第三方支付平台上进行交易。

第三方支付具有以下两个功能。

- 兑换CarCoin。用户可以通过在第三方支付平台的账号,与趣快出行官方账号交易,购买CarCoin。

□ 提现。用户可以通过在第三方支付平台的账号，与趣快出行官方账号交易，卖出CarCoin。

9.2.3 系统总体设计

趣快出行系统是一个基于区块链的去中心化的网约车平台，在设计上要符合去中心化这一基本特性。本节将从业务逻辑设计和系统架构设计两方面介绍趣快出行系统的设计。

1. 业务逻辑设计

趣快出行业务逻辑主要分为两部分：行程业务逻辑与支付业务逻辑。

(1) 行程业务逻辑

在行程业务逻辑中，乘客的状态有4种：空闲、约车、等待接驾、行程中；司机的状态有4种：离线、在线、接客中、行程中。行程业务逻辑设计如下。

- 1) 乘客输入出发地、目的地等基本信息后，发起约车请求。此时乘客的状态从空闲变为约车。
- 2) 系统根据订单撮合逻辑，将订单发送给合适的处于在线状态的司机。司机选择一条订单。
- 3) 乘客收到预付款请求，支付预付款。支付成功后，乘客状态从约车变为等待接驾，司机状态变为接客中。
- 4) 司机接到乘客后，司机确认接到乘客。此时，乘客与司机的状态都变为行程中。
- 5) 到达目的地后，司机点击结束订单。此时，乘客的状态变为初始空闲状态，司机的状态变为初始在线状态。一个完整的订单结束。

(2) 支付业务逻辑

支付业务逻辑分为两部分：支付预付款和账单结算。

1) 支付预付款。首先，乘客收到系统的支付预付款请求。乘客点击付款后，就会在第三方支付平台上将人民币支付给趣快出行平台账户。当平台账号收到人民币后，就会将等额的CarCoin充值到乘客区块链账户上。然后，乘客使用CarCoin支付预付款。当然，如果乘客CarCoin余额足够支付预付款，乘客可以选择直接使用CarCoin支付。

2) 账单结算。账单结算部分比较简单。当行程结束后，支付合约根据订单的实际费用以及预付款结算车费，将实际车费支付给司机，将多余的预付款返还给乘客。

业务逻辑流程图如图9.7所示。

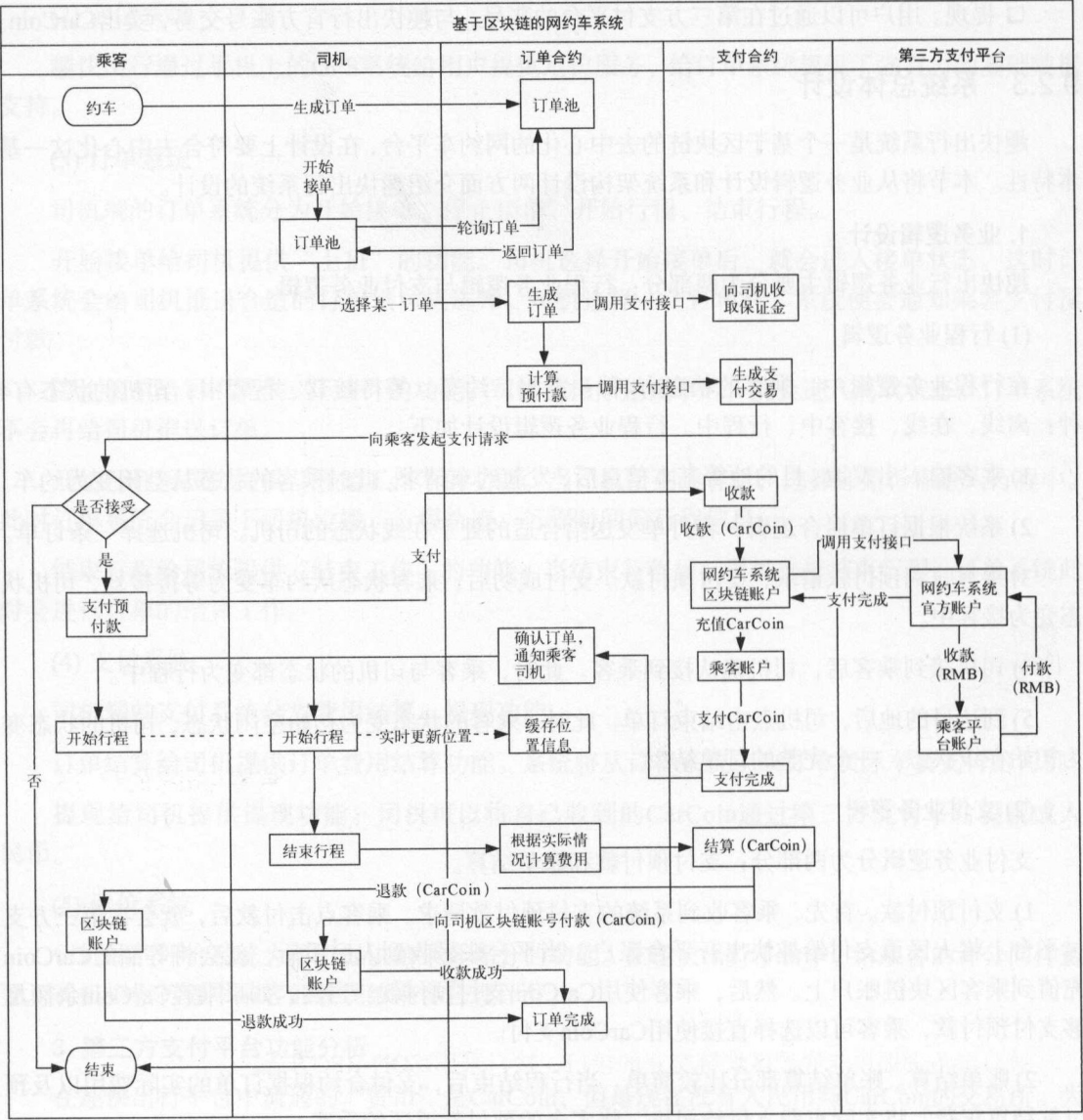


图9.7 趣快出行业务逻辑图

2. 系统架构设计

系统架构图如图9.8所示。

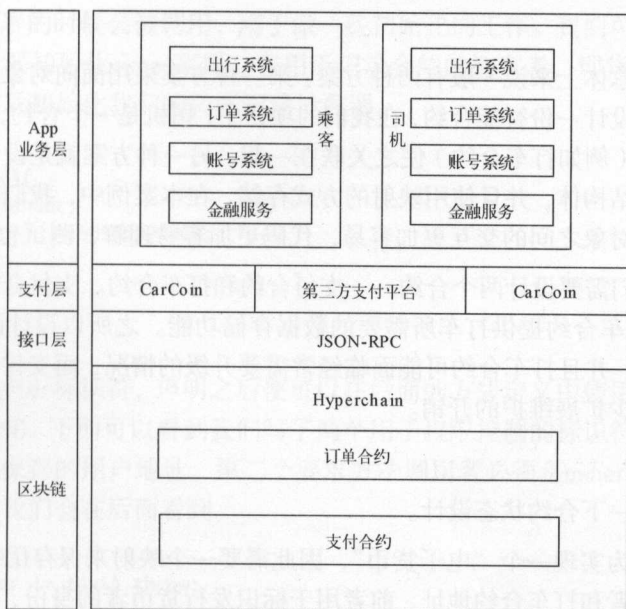


图9.8 趣快出行系统架构图

趣快出行系统分为4层。

(1) 业务层

业务层负责业务逻辑的处理，通过App形式实现。在业务层实现了乘客与司机的身份系统、定位系统、订单系统、支付系统，用户可以通过App使用这些功能。

(2) 支付层

支付层负责提供支付相关的接口，分为两部分，一部分是CarCoin支付相关接口，包括支付等；另一部分是第三方支付相关接口，包括兑换CarCoin、提现等。

(3) 接口层

接口层负责提供与底层区块链交互的接口，接口采用JSON-RPC方式。

(4) 区块链

区块链层提供区块链平台。在区块链中，部署了趣快出行项目的两个智能合约——订单合约和支付合约。

9.2.4 智能合约设计

趣快出行系统使用智能合约来实现主体业务逻辑。智能合约既是程序逻辑的主体，也是数据存储的主体，本节介绍了趣快出行系统中智能合约的设计。

1. 概念设计

智能合约的设计总体上来说一般有两种方案：第一种方案采用面向对象的思想，对于每一个项目中的对象都对应设计一份智能合约，在我们的项目中，司机是一个合约，乘客也是一个合约，然后通过第三个合约（例如打车合约）使之关联在一起。另一种方案就是设计一个合约，不同的对象表现为合约中的结构体，并且使用映射的方式存储。在本案例中，我们使用第二种方案，相对来说，第二种方案对象之间的交互更加容易，代码更加容易理解，测试也较为简单。

总体上来说，我们需要设计两个合约——支付合约和打车合约。支付合约实现“电子现金”CarCoin的功能，而打车合约提供打车所需要的数据存储功能。之所以设计两个合约，是因为它们之间的耦合度较低，并且打车合约可能面临经常需要升级的情况，而支付合约则相对稳定，因此这样的设计可以减少扩展维护的开销。

2. 支付合约设计

首先我们来介绍一下合约状态设计。

支付合约的功能为实现一个“电子货币”，因此需要一个映射来保存用户的余额。同时我们需要保存合约的拥有者和打车合约地址，前者用于标识发行货币者的身份，并且对充值、提现进行权限控制，相当于一个管理员的角色。后者则用于转账的权限控制，使得只有打车合约能够调用预付款、结账等涉及转账的方法。

在实际生活中，货币的最小单位并不是常用单位，例如人民币最小单位是“分”，而常用单位却是“元”。为了能和人民币进行对应，我们设定CarCoin的最小单位也是分，因此使用一个int类型来保存的时候，1代表1分。

```
address owner; //合约拥有者，“货币”发行者
address taxi; //打车合约
mapping (address => int) balances; //用户余额
```

除此之外，我们还需要一个struct结构体来保存转账的记录，它有4个属性：付款方、收款方、金额、备注。备注信息用来表明转账的意图。同时还有一个计数器记录这些记录的总数，因为我们的转账记录是按照编号顺序存储的，这样做是为了节省开销。最后我们需要一个映射，使用编号来映射到具体的结构体中。

```
struct record{
    address from; //付款方
    address to; //收款方
    int value; //金额
    string comment; //转账意图
}
uint counter; //计数器
mapping (uint => record) records; //编号映射到结构体
```

接下来我们详细介绍合约方法设计。

(1) 构造方法

每个合约都有一个默认的构造函数，这和面向对象的程序设计（例如C++）中的构造函数是

一样的，它在合约部署的时候会被调用，用于做一些初始化的工作。我们可以对它进行重写。在我们的构造函数中，要初始化owner字段，它用于记录合约的拥有者，即货币发行者，之后设定发行的总货币量，最后初始化我们的转账记录计数器。

```
//构造函数
function CarCoin(){
    owner = msg.sender;
    balances[owner] = 1000000000000;
    counter = 1;
}
```

(2) 权限控制符

Solidity中有modifier标识符，声明之后便可以在后面的方法定义中使用，这样可以减少代码中重复的权限控制语句。下面可以看到我们写了两个用于权限控制的标识符，第一个要求方法调用者必须是taxi字段保存的用户地址，第二个要求方法调用者必须是owner字段保存的合约拥有者。它们的具体用法我们会在后面看到。

```
modifier onlyTaxi(){
    if (msg.sender != taxi) throw;
    _;
}

modifier onlyOwner(){
    if (msg.sender != owner) throw;
    _;
}
```

(3) 注册函数

由于打车合约可能被重新部署以完成升级、维护等操作，因此我们的taxi字段应该可以动态改变。这个函数需要使用刚才设定的标识符来进行权限控制，使之只能被合约拥有者调用。

```
function exeOnce(address addr) onlyOwner{
    taxi = addr;
}
```

(4) 记录函数

根据前面的设定，任何转账行为都应该有对应的记录，下面的函数就会将记录存储起来。

```
//记录函数
function Transfer(address from, address to, int value, string comment) private{
    records[counter].from = from;
    records[counter].to = to;
    records[counter].value = value;
    records[counter].comment = comment;
    counter++;
}
```

(5) 转账函数

虽然我们实现的是一个“电子货币”，但是我们只允许在打车流程中进行流通，而不允许用户随意自由转账。根据场景我们将转账函数细分为4个：预付款、结账、收取违约金、充值/提现。前三者只允许打车合约调用，最后一个只允许合约拥有者调用。可以看到，我们之前定义的标识符被用在了函数声明中，用来进行权限控制。顺便一提，充值/提现使用的是同一个函数，因为我们只要设定金额为负数，就实现了提现。

//预付款函数

```
function prepay(address client, int preFee) onlyTaxi
    returns(bool success){
    balances[client] -= preFee;
    balances[owner] += preFee;
    Transfer(client, this, preFee, "prepay");
    return true;
}
```

//结账函数

```
function confirm(address client, address driver, int preFee,
    int finalFee) onlyTaxi returns(bool success){
    int remain = preFee - finalFee;
    balances[owner] -= preFee;
    balances[client] += remain;
    balances[driver] += finalFee;
    Transfer(this, client, remain, "remain fee");
    Transfer(this, driver, finalFee, "final fee");
    return true;
}
```

//违约金函数

```
function penalty(address from, address to, int amount)
    onlyTaxi returns(bool){
    balances[from] -= amount;
    balances[to] += amount;
}
}
```

//充值/提现函数

```
function recharge(address addr, int amount) onlyOwner
    returns(bool){
    if (balances[owner] < amount){
        return false;
    }
    balances[addr] += amount;
    balances[owner] -= amount;
    Transfer(this, addr, amount, "recharge");
    return true;
}
```

(6) 查询函数

当然，我们要能够查询用户的余额，顺便可以查看合约拥有者到底是谁。

//查询余额

```
function getBalance(address addr) returns(int){
    return balances[addr];
}
```

```

}
//查询合约所有者
function getOwner() returns(address){
    return owner;
}

```

3. 打车合约设计

打车合约要完成一个完整的打车流程，其主要对象有：订单、司机、乘客和评价。其中乘客并没有单独抽象出来，因为在打车的场景中，并不需要具体的乘客信息，只需要乘客的联系方式以及地址即可，而它们都被包含在了订单对象中，另外单独列成结构体的是乘客的位置信息，它只是为了使得在打车这个场景中某些逻辑更加合理罢了，并不是必需的。

关于我们的数据结构类型，由于Solidity并不支持浮点数的计算，因此需要用整数来模拟。前面对于CarCoin的设定便是如此。这里我们还要处理有关距离的数据，经纬度在这里保留了小数点后六位，也就是说，120°在这里存放的实际形式为120000000，而根据大致的计算，其中1表示的是0.1米。至于时间，我们采用的是Linux时间戳，其值为1970年1月1日到当前的秒数。评分则采用[0, 5000]的数值，用来表示5分制，即可以保留小数点后3位。

我们首先列出订单对象，它包含了如下属性：订单编号、乘客地址、司机地址、起终点经纬度、起终点地名、距离、时间、费用、状态、乘客信息和司机信息。这些信息的含义也十分好理解，就如其字面意思一样。同时有一个计数器表明当前的订单数，一个映射将编号映射到具体的订单结构中。

```

struct Order{
    uint id; //订单编号
    address passenger; //乘客地址
    address driver; //司机地址
    int s_x; //起点经度
    int s_y; //起点纬度
    int d_x; //终点经度
    int d_y; //终点纬度
    string sName; //起点地名
    string dName; //终点地名
    int distance; //起终点直线距离
    int preFee; //预付款额
    int actFee; //里程费
    int actFeeTime; //时长费
    uint startTime; //订单提交时间 UNIX标准时间
    uint pickTime; //接客时间
    uint endTime; //结束时间
    int state; //订单状态 1待分配 2已被抢 3订单完成 4订单终止
    string passInfo; //乘客个人信息
    string drivInfo0; //司机个人信息
    string drivInfo1;
    string drivInfo2;
}

uint counterOrderIndex; //下一个空的订单序号
mapping (uint => Order) orders;

```

司机结构保存了有关司机的信息，其属性有：经纬度、是否接单、自身地址、信息、订单池、上一次经纬度。需要说明的是“上一次经纬度”这个属性，它用于进行实时计费，是一个辅助属性。另外，由于订单并不需要一一对应到某一个司机或者乘客上（当某一单结束之后，新的一单开始），所以只需要编号作为映射的键值即可，而司机结构和具体的司机是一一对应的，并且司机内部仍然需要一个编号，于是我们就看到了双重映射，司机的地址映射到了一个编号上，再由这个编号映射到具体的结构体上。

```
mapping (address => uint) driverIndexs; //给每个司机分配一个内部的序号
uint counterDriverIndex; //下一个空的司机序号（当前司机数量+1）
struct Driver{
    int cor_x; //经度
    int cor_y; //纬度
    bool state; //true 表示接单中 false 表示休息中
    address name; //司机地址
    string info0;
    string info1;
    string info2;
    uint counterOrder; //司机当前可接订单数
    uint[8] orderPool; //司机可接订单池
    int last_x; //上一次经度
    int last_y; //上一次纬度
}
mapping (uint => Driver) drivers; //使用序号去寻找司机的信息
```

评价结构较为简单，它的属性有：总评价数、平均评价、单次分数、单次评价。其中，单次分数和评价都是按顺序存放的。同时也有一个映射将司机地址映射到评价结构中。

```
struct Judgement{
    int total; //总评价数
    int avgScore; //平均分
    mapping (int => int) score; //单次分数
    mapping (int => string) comment; //单次评价
}
mapping (address => Judgement) driverJudgements;
```

乘客位置结构，不再赘述。

```
struct passengerPosition{
    int x;
    int y;
}
mapping (address => passengerPosition) passPos;
```

最后，还有一些独立的结构，首先是支付合约结构，这是一种合约之间互相调用的方法，和C++中的类十分相像，调用某个合约的方法只需要使用“变量.方法名”的形式即可。然后我们还有两个映射，分别将乘客和司机映射到某个订单上，表达的是当前乘客、司机所处的订单。正如前面所说，它会因为旧订单的结束、新订单的开始而被更新。之后是两个映射，用来存储用户的当前状态，状态是表明用户所处哪个阶段的重要标识，在合约方法设计中我们会详述。最后一个映射表明附近的司机。


```
CarCoin carcoin;
```

```
//每个乘客对应到某个订单
mapping (address => uint) passengerToOrder;
//每个司机对应到某个订单
mapping (address => uint) driverToOrder;
//乘客状态
mapping (address => uint) passengerStates;
//司机状态
mapping (address => uint) driverStates;
//附近的司机
mapping (address => uint[5]) passengerNearDrivers;
```

接下来我们来详细介绍合约方法设计。

(1) 构造函数

不同于默认的构造函数，这里使用了带参构造函数，参数是支付合约的地址。因为在打车流程中需要多次使用支付合约的方法。除此之外，就是对订单计数器 and 司机计数器进行初始化。

```
function Taxi(address cc){
    counterDriverIndex = 1;
    counterOrderIndex = 1;
    carcoin = CarCoin(cc);
}
```

(2) 私有函数

如同C++一样，我们可以定义一些私有函数，它们只能在内部使用。这里我们有4个私有函数。第一个是开根函数，Solidity并没有现成的对整数进行开根的函数，于是我们需要自己定义。有关整数开根的问题，网上有很多讨论，它们可以获得精度较高的整数平方根。其次是距离计算函数，用的是两点距离公式，在本案例中其实是有所偏差的，因为根据两点之间的经纬度计算实际的距离有一套计算公式，要考虑地球半径以及经纬度位置等，而我们将球面的距离公式简化成了平面的距离公式，因为Solidity并不适合进行复杂的数学运算，加之小范围的距离误差并无多大影响，故采用此下策。第三是订单分配函数，也采用了最简单的方案，即处在一定范围内的司机都被分配。最后是计算预付款函数，其原理是计算两点的直角边距离，然后乘上一个系数。

```
function sqrt(int x) private returns (int){
    if(x < 0)
        x = - x;
    int z = (x + 1) / 2;
    int y = x;
    while (z < y){
        y = z;
        z = (x / z + z) / 2;
    }
    return y;
}
```

```
function calculateDistance(int x0, int x1, int y0, int y1)
```

```

        private returns(int){
            int tempX = x0 - x1;
            int tempY = y0 - y1;
            return sqrt(tempX*tempX + tempY*tempY);
        }

function driverSelction(int x, int y, uint orderIndex)
    private returns(bool){
        uint i;
        uint j;
        int threshold = 50000; //阈值, 当距离小于该值之后则派单, 数值可调整
        int temp;
        uint maxOrder = 8; //司机可抢的最大订单数量
        bool flag = false;
        for (i=1; i<counterDriverIndex; ++i){
            if (drivers[i].state && driverStates[drivers[i].name] == 0){
                temp = calculateDistance(x, drivers[i].cor_x, y,
                    drivers[i].cor_y);
                if (temp < threshold){
                    //找到订单池中的空位
                    for(j=0; j<maxOrder; ++j){
                        if(orders[drivers[i].orderPool[j]].state != 1){
                            flag = true;
                            drivers[i].orderPool[j] = orderIndex;
                            break;
                        }
                    }
                }
            }
        }
        return flag;
    }

function calculatePreFee(int s_x, int s_y, int d_x, int d_y)
    private returns(int){
        int tempX = s_x - d_x;
        int tempY = s_y - d_y;
        if (tempX < 0){
            tempX = -tempX;
        }
        if (tempY < 0){
            tempY = -tempY;
        }
        return ((tempX + tempY) * unitPrice) / 2 * 3 / 100;
    }
}

```

(3) 乘客提交订单

正如其字面意思, 乘客通过该函数提交订单请求, 合约会分配一个新的订单编号, 并且将信息写入, 同时分配给司机。如果乘客余额不足, 或是没有司机, 又或是状态不正确 (例如上一单还未结束, 新的订单就不会被分配), 都会出现提交失败。提交后会返回订单编号, 通过这个编号, 乘客可以很方便地查询到订单的具体信息。如果提交成功, 则会设置对应的状态, 进入等待

司机抢单的阶段。

```
function passengerSubmitOrder(int s_x, int s_y, int d_x, int
    d_y, uint time, string passInfo, string sName, string
    dName) returns(uint){
```

```
    //乘客账户余额必须是正数
```

```
    if(carcoin.getBalance(msg.sender) < 0){
        return 0;
    }
```

```
    //乘客必须处于挂起状态才能抢单
```

```
    if(passengerStates[msg.sender] != 0){
        return 0;
    }
```

```
    if (counterDriverIndex <= 1){ //没有司机
        return 0;
    }
```

```
    //创建新的订单
```

```
    passengerToOrder[msg.sender] = counterOrderIndex;
    orders[counterOrderIndex].id = counterOrderIndex;
    orders[counterOrderIndex].passenger = msg.sender;
    orders[counterOrderIndex].driver = 0x0;
```

```
    orders[counterOrderIndex].s_x = s_x;
```

```
    orders[counterOrderIndex].s_y = s_y;
```

```
    orders[counterOrderIndex].d_x = d_x;
```

```
    orders[counterOrderIndex].d_y = d_y;
```

```
    orders[counterOrderIndex].distance = 0;
```

```
    orders[counterOrderIndex].preFee = penaltyPrice +
```

```
        calculatePreFee(s_x, s_y, d_x, d_y);
```

```
    orders[counterOrderIndex].actFee = 0;
```

```
    orders[counterOrderIndex].actFeeTime = 0;
```

```
    orders[counterOrderIndex].startTime = time;
```

```
    orders[counterOrderIndex].state = 1;
```

```
    orders[counterOrderIndex].passInfo = passInfo;
```

```
    orders[counterOrderIndex].sName = sName;
```

```
    orders[counterOrderIndex].dName = dName;
```

```
    counterOrderIndex++;
```

```
    passengerStates[msg.sender] = 1; //乘客订单分配中
```

```
    if(!driverSelction(s_x, s_y, counterOrderIndex-1)){
```

```
        orders[counterOrderIndex-1].state = 4;
```

```
        passengerStates[msg.sender] = 0;
```

```
        return 0;
    }
```

```
    return counterOrderIndex-1;
```

(4) 司机抢单

我们采取的是司机抢单的模式，通过这个函数就可以完成这个操作，它会检查抢单的条件，如果成功则会设置对应的状态，进入等待乘客预付款的阶段。

```

function driverCompetOrder(uint orderIndex) returns(bool){
    if(driverIndex[msg.sender] == 0){//司机没有注册
        return false;
    }
    if(driverStates[msg.sender] != 0){//司机不在挂起状态
        return false;
    }
    if(orders[orderIndex].state != 1){//抢单失败
        return false;
    }
    orders[orderIndex].state = 2;
    orders[orderIndex].driver = msg.sender;
    orders[orderIndex].drivInfo0 =
        drivers[driverIndex[msg.sender]].info0;
    orders[orderIndex].drivInfo1 =
        drivers[driverIndex[msg.sender]].info1;
    orders[orderIndex].drivInfo2 =
        drivers[driverIndex[msg.sender]].info2;
    passengerStates[orders[orderIndex].passenger] = 2; //乘客待付款
    driverStates[msg.sender] = 1; //司机已接单
    driverToOrder[msg.sender] = orderIndex;
    //初始化司机上一次位置
    drivers[driverIndex[msg.sender]].last_x =
        orders[orderIndex].s_x;
    drivers[driverIndex[msg.sender]].last_y =
        orders[orderIndex].s_y;
    return true;
}

```

(5) 乘客预付款

我们采取预付款的模式，这里的预付款就是调用支付合约的接口，然后对状态进行检查，如果成功，则会设置对应的状态，并且进入等待司机接客的阶段。

```

function passengerPrepayFee() returns(bool){
    uint orderIndex = passengerToOrder[msg.sender];
    address driver = orders[orderIndex].driver;

    //乘客不是待付款，或者订单不是已被抢
    if (passengerStates[msg.sender] != 2 ||
        orders[orderIndex].state != 2){
        return false;
    }

    //付款过程，确定款项已经进入合约账户
    if (carcoin.prepay(msg.sender, orders[orderIndex].preFee)){
        passengerStates[msg.sender] = 3;
        driverStates[driver] = 2;
        return true;
    } else {
        //....
        orders[orderIndex].state = 4;
        passengerStates[msg.sender] = 0;
        driverStates[driver] = 0;
    }
}

```



```

        return false;
    }
}

```

(6) 司机接客

当司机接到乘客的时候，调用该函数来进行状态检查和设置，这里我们还进行了防作弊的检验，即只有司机和乘客的当前位置足够接近的时候才能调用成功，防止司机进行欺诈。这样的自动化判断是十分必要的，因为这是一个完全无人值守的系统，严苛的检查有利于减少纠纷。当调用成功之后，就会进入行程中的状态。

```

function driverPickUpPassenger(int x, int y, uint time)
    returns(bool){
    uint orderIndex = driverToOrder[msg.sender];
    address passenger = orders[orderIndex].passenger;

    //状态检查
    if (driverStates[msg.sender] != 2 ||
        passengerStates[passenger] != 3 ||
        orders[orderIndex].state != 2){
        return false;
    }

    int passX = passPos[passenger].x;
    int passY = passPos[passenger].y;
    int threshold = 20000;

    if (calculateDistance(x, passX, y, passY) > threshold){
        return false;
    }

    drivers[driverIndexs[msg.sender]].last_x = x;
    drivers[driverIndexs[msg.sender]].last_y = y;
    orders[orderIndex].pickTime = time;

    passengerStates[passenger] = 4;
    driverStates[msg.sender] = 3;
    return true;
}

```

(7) 实时计费

实现实时计费的方法为，司机在行程中不断调用该函数，并且传入当前的位置，进行分段计费，当分段分得足够细的时候，我们就可以用每段的直线距离总和来近似估计行程总长，达到实时计费的效果。

```

function driverCalculateActFee(int cur_x, int cur_y) returns(int){
    uint orderIndex = driverToOrder[msg.sender];
    uint driverIndex = driverIndexs[msg.sender];
    int distance;
    address passenger = orders[orderIndex].passenger;

```

```

//状态检查
if (driverStates[msg.sender] != 3 ||
    passengerStates[passenger] != 4 ||
    orders[orderIndex].state != 2){
    return 0;
}

distance = calculateDistance(cur_x,
    drivers[driverindex].last_x, cur_y,
    drivers[driverindex].last_y);
orders[orderIndex].distance += distance;
orders[orderIndex].actFee += distance * unitPrice / 100;
drivers[driverindex].cor_x = cur_x;
drivers[driverindex].cor_y = cur_y;
drivers[driverindex].last_x = cur_x;
drivers[driverindex].last_y = cur_y;
return orders[orderIndex].actFee;
}

```

(8) 完成订单

司机在乘客下车之后可以使用此函数来完成订单，系统会根据实际费用和预付款额来进行结算，并且调用支付合约的接口完成支付。

```

function driverFinishOrder(uint time) returns(bool){
    uint orderIndex = driverToOrder[msg.sender];
    address passenger = orders[orderIndex].passenger;
    //司机不是行程中，订单不是已被抢
    if (driverStates[msg.sender] != 3 ||
        passengerStates[passenger] != 4 ||
        orders[orderIndex].state != 2){
        return false;
    }
    if (time < orders[orderIndex].pickTime){
        time = orders[orderIndex].pickTime;
    }
    orders[orderIndex].actFeeTime = (int)(time -
        orders[orderIndex].pickTime) * unitPriceTime;
    int preFee = orders[orderIndex].preFee;
    int finalFee = orders[orderIndex].actFee +
        orders[orderIndex].actFeeTime;
    if (finalFee > preFee){
        finalFee = preFee;
    }
    orders[orderIndex].actFee = finalFee -
        orders[orderIndex].actFeeTime;
}

//支付
if (carcoin.confirm(passenger, msg.sender, preFee,
    finalFee)){
    orders[orderIndex].state = 3;
    orders[orderIndex].endTime = time;
    passengerStates[passenger] = 0;
    driverStates[msg.sender] = 0;
}

```

```

        return true;
    } else {
        //....
        passengerStates[msg.sender] = 0;
        driverStates[msg.sender] = 0;
        orders[orderIndex].state = 4;
        return false;
    }
}

```

(9) 取消函数

订单取消是实际情况中经常遇到的，我们提供了统一的接口，系统会自动判断当前的状态是否可以取消，并且进行设置。乘客调用对应乘客的取消函数，司机调用对应司机的取消函数。

```

function passengerCancelOrder(bool isPenalty) returns(bool){
    uint orderIndex = passengerToOrder[msg.sender];
    address driver = orders[orderIndex].driver;

    //乘客在司机接单前取消订单，没有任何惩罚
    if (passengerStates[msg.sender] == 1 &&
        orders[orderIndex].state == 1){
        passengerStates[msg.sender] = 0;
        orders[orderIndex].state = 4;
        return true;
    }

    //乘客在司机接单后、自己预付款前取消订单，没有惩罚
    if (passengerStates[msg.sender] == 2 &&
        driverStates[driver] == 1 && orders[orderIndex].state
        == 2){
        passengerStates[msg.sender] = 0;
        driverStates[driver] = 0;
        orders[orderIndex].state = 4;
        return true;
    }

    //乘客在预付款后、等待司机接客时取消订单
    if (passengerStates[msg.sender] == 3 &&
        driverStates[driver] == 2 && orders[orderIndex].state
        == 2){
        //退还预付款
        if (!carcoin.confirm(msg.sender, driver,
            orders[orderIndex].preFee, 0)){
            return false;
        }
        //违约金
        if (isPenalty){
            carcoin.penalty(msg.sender, driver, penaltyPrice);
        }
        passengerStates[msg.sender] = 0;
        driverStates[driver] = 0;
        orders[orderIndex].state = 4;
        return true;
    }
}

```

```

    }
    return false;
}

function driverCancelOrder() returns(bool){
    uint orderIndex = driverToOrder[msg.sender];
    address passenger = orders[orderIndex].passenger;

    //司机在乘客预付款前取消
    if (driverStates[msg.sender] == 1 &&
        passengerStates[passenger] == 2 &&
        orders[orderIndex].state == 2){
        passengerStates[passenger] = 0;
        driverStates[msg.sender] = 0;
        orders[orderIndex].state = 4;
        return true;
    }
    //司机在乘客预付款后取消, 有违约金
    if (driverStates[msg.sender] == 2 &&
        passengerStates[passenger] == 3 &&
        orders[orderIndex].state == 2){
        //退还预付款
        if (!carcoin.confirm(passenger, msg.sender,
            orders[orderIndex].preFee, 0)){
            return false;
        }
        carcoin.penalty(msg.sender, passenger, penaltyPrice);
        passengerStates[passenger] = 0;
        driverStates[msg.sender] = 0;
        orders[orderIndex].state = 4;
        return true;
    }
    return false;
}
}

```

(10) 乘客评价

乘客在完成一笔订单之后可以进行评价, 评价完成之后会解除绑定, 乘客就无法再次评价了。

```

function passengerJudge(int score, string comment)
    returns(bool){
    uint orderIndex = passengerToOrder[msg.sender];
    address driver = orders[orderIndex].driver;
    int total = driverJudgements[driver].total;

    if (orderIndex == 0){
        return false;
    }

    passengerToOrder[msg.sender] = 0; //解除绑定
    if (score > 5000)
        score = 5000;
    if (score < 0)

```



```

        score = 0;
        driverJudgements[driver].avgScore =
            (driverJudgements[driver].avgScore * total + score) /
            (total + 1);
        driverJudgements[driver].total += 1;
        total++;
        driverJudgements[driver].score[total] = score;
        driverJudgements[driver].comment[total] = comment;
        return true;
    }

```

(11) 司机注册

司机在合约内部是有一个编号的，新的司机并不会直接分配出一个司机结构体，因此需要调用该函数进行“注册”。

```

function newDriverRegister(string info0, string info1, string
    info2) returns(uint){
    if (driverIndexs[msg.sender] > 0){//已经注册
        return driverIndexs[msg.sender];
    }
    driverIndexs[msg.sender] = counterDriverIndex;
    drivers[counterDriverIndex].state = false;
    drivers[counterDriverIndex].name = msg.sender;
    drivers[counterDriverIndex].cor_x = 0;
    drivers[counterDriverIndex].cor_y = 0;
    drivers[counterDriverIndex].info0 = info0;
    drivers[counterDriverIndex].info1 = info1;
    drivers[counterDriverIndex].info2 = info2;
    drivers[counterDriverIndex].counterOrder = 0;
    driverStates[msg.sender] = 0;
    counterDriverIndex++;
    return counterDriverIndex - 1;
}

```

(12) 查询函数

由于智能合约的特殊性，Solidity并不支持主动式的通知，因此所有信息都需要客户端来主动进行查询，因此需要大量的查询函数来支持查询。这些查询函数的结构非常类似，都是返回某些结构体的具体内容罢了。下面仅以查询订单信息为例进行展示，详情不再赘述，读者可以参看合约源代码。

```

function getOrderInfo0(uint orderIndex) returns(uint id,
    address passenger, int s_x, int s_y, int d_x, int d_y,
    int distance, int preFee, uint startTime, string
    passInfo){
    id = orders[orderIndex].id;
    passenger = orders[orderIndex].passenger;
    s_x = orders[orderIndex].s_x;
    s_y = orders[orderIndex].s_y;
    d_x = orders[orderIndex].d_x;
    d_y = orders[orderIndex].d_y;
}

```

```

distance = orders[orderIndex].distance;
preFee = orders[orderIndex].preFee;
startTime = orders[orderIndex].startTime;
passInfo = orders[orderIndex].passInfo;
}

```

9.2.5 系统实现与部署

趣快出行系统分为App前端和区块链后台两部分。App前端直接下载安装便可使用，这里不再赘述。本节将介绍趣快出行系统区块链后台的部署。

1. 系统部署图

系统部署图如图9.9所示。

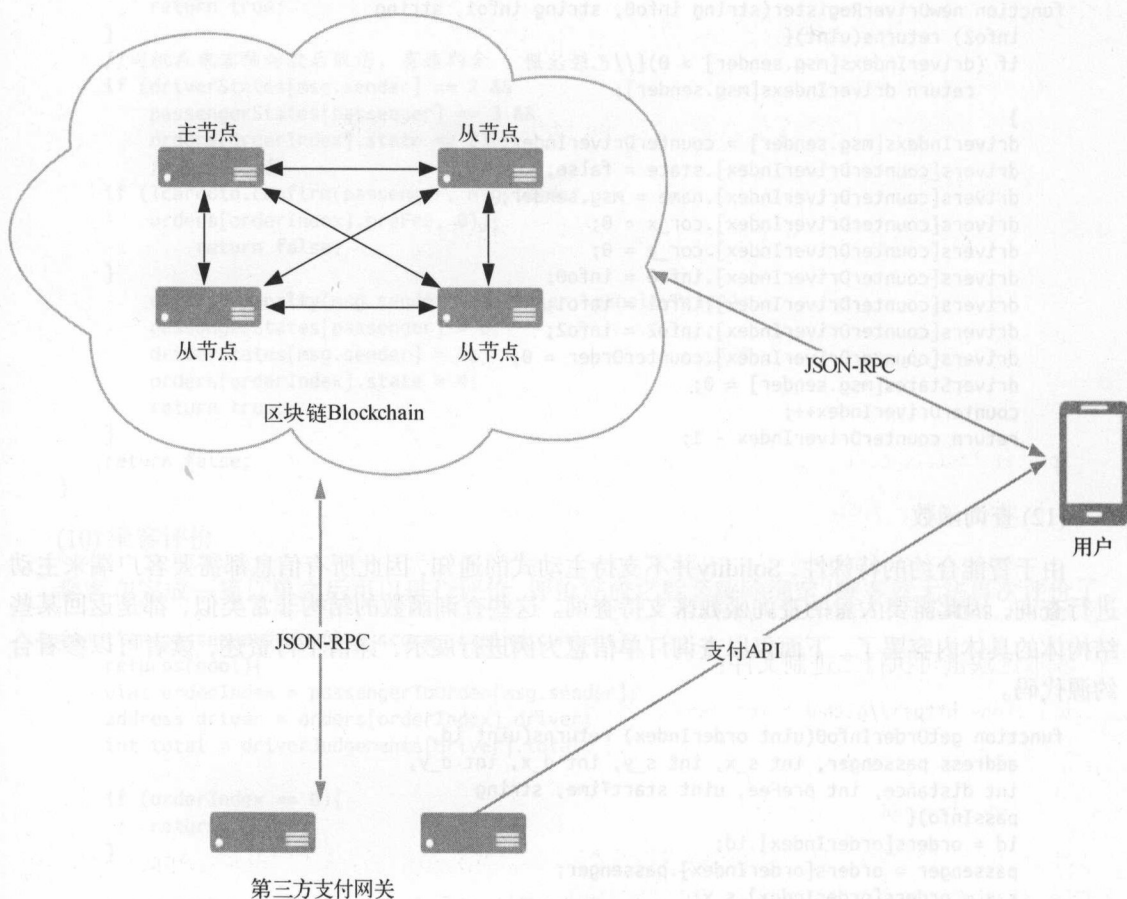


图9.9 系统部署图

本案例采用Hyperchain底层区块链平台，趣快出行、第三方支付平台等合作方作为区块链节点加入Hyperchain。乘客和司机通过App接入Hyperchain。第三方支付平台的支付服务器作为系统的支付网关，与Hyperchain连接；App则通过第三方支付API连接支付网关。

2. 系统部署环境

● 硬件环境

区块链：阿里云服务器、双核CPU、4GB内存、500GB可用存储空间

App：iPhone

第三方支付网关：阿里云服务器、双核CPU、4GB内存、200GB可用存储空间

● 软件环境

区块链：Linux操作系统（Ubuntu 14.04）；Go语言环境

App：iOS 8.0以上

第三方支付网关：Linux操作系统（Ubuntu 14.04）；Go语言环境

3. 区块链环境搭建

对于一台空的服务器，可按照以下步骤搭建区块链环境。

安装Git：

```
apt-get update
apt-get install git
```

安装Curl：

```
apt-get install curl
```

安装配置Go语言环境：

```
curl -O https://storage.googleapis.com/golang/go1.8.1.linux-amd64.tar.gz
tar -C /usr/local -xzf go1.8.1.linux-amd64.tar.gz
export PATH=$PATH:/usr/local/go/bin
```

获取区块链可执行二进制文件：

```
git clone https://github.com/trakel-project/trakelchain.git
```

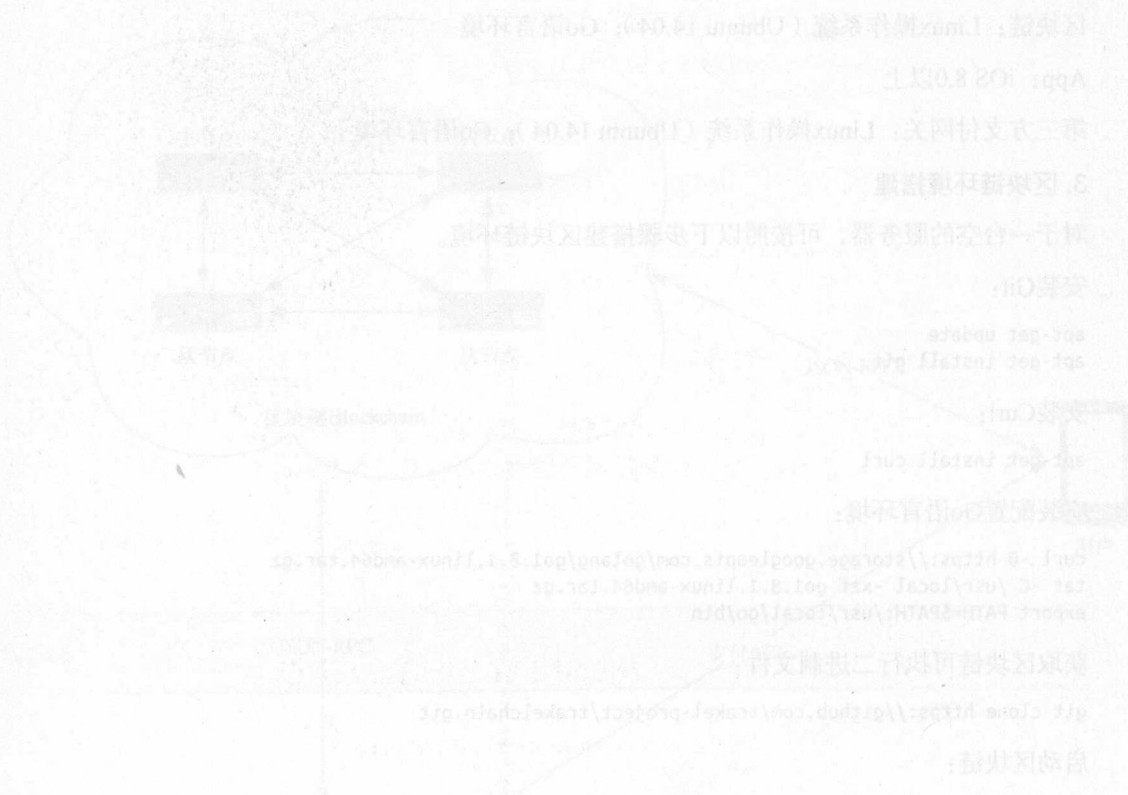
启动区块链：

```
cd trakelchain && ./start.sh
```

注意，以上环境搭建是区块链环境搭建步骤，不包含第三方支付网关以及iOS App。trakelchain已部署趣快出行所需要的智能合约，可直接调用。

9.3 本章小结

本章给出了两个基于Hyperchain的企业级区块链应用项目案例，每个案例的介绍均包括项目简介、系统功能分析、系统总体设计、智能合约设计、系统实现和部署等部分。可以看到，利用Hyperchain可以构建功能完备、技术领先、符合企业级要求的区块链应用。读者可对照本章内容，通过Hyperchain提供的完善的开发接口，对区块链应用开发进行深入的学习和实践。





微信连接



回复“软件开发”查看相关图书



微博连接

关注 @图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I: 218139230

图灵读者官方群II: 164939616

图灵社区
iTuring.cn

在线出版, 电子书, 《码农》杂志, 图灵访谈

贾圣林

浙江大学互联网金融
研究院院长

本书由浙江大学和趣链科技联合撰写，融合了浙江大学的理论研究成果和趣链科技的实际开发经验，是一本实践性非常强的区块链技术图书。

杨小虎

浙江大学软件学院
常务副院长

本书从技术原理、市场发展、政府规划、应用场景和平台对比等角度对区块链进行了全景分析，对三个主流区块链平台进行了深入介绍，配有动手开发指南，并提供了实际项目案例说明和源代码，是一本不可多得的理论与实践相结合的区块链技术图书。

宋士正

浙商银行信息科技部
总经理

本书非常务实，干货满满，给读者呈现了当前区块链技术发展的新动态，所提供的案例内容翔实，其背后必然有脚踏实地做技术的“实战派”团队支持。相信本书会对有志于从事区块链技术研究和应用开发的人员有帮助。

史晨阳

中国光大银行信息
科技部副总经理

本书按照区块链基础知识、开源平台、企业级平台和开发案例的顺序介绍了区块链技术的基本原理和开发技术，并包含多个案例，非常适合相关技术人员由浅入深地学习区块链技术。

图灵社区: iTuring.cn

反馈/投稿/推荐邮箱: contact@turingbook.com

读者热线: (010) 51095186-600

分类建议 计算机/软件开发/程序设计

人民邮电出版社网址: www.ptpress.com.cn



ISBN 978-7-115-47179-6



ISBN 978-7-115-47179-6

定价: 69.00元